APPLIED ROBOT VISION WITH I-SEE-BYTES

İbrahim Cem Baykal

APPLIED ROBOT VISION WITH I-SEE-BYTES

İBRAHİM CEM BAYKAL

1.1st EDITION
April 2025

A publication of Otoidrak Software Ltd. ISBN:

INDEX

Preface

Every graduate student who had to deal with the OpenCV must have realized that it is a horribly impractical library to write programs with. First of all, it has almost no support for the GUI widgets such as the text edit boxes, buttons, list boxes, menus etc. It doesn't create a window frame for your application; it just shows pictures. The Python version is easier to use but it is very slow compared to the C++ version and if you don't want to be an academician but work in the industry, you need to learn C++ because the industry uses it for real-time computer vision products and Python is too slow for that.

The fact that there are frustrating number of different objects and structures in the OpenCV library pushed me to think about this issue, and over the years, I started to design a way to keep many different variable types and data structure definitions in a compressed form. The I-See-Bytes library emerged as a result of this effort. The main object of this library, the ICBYTES class (pronounced I-See-Bytes), is used like a variable of Python. Not only can it hold int, float, double, etc. type variables but also complex data structures such as matrices and vector lists and it can alter its contents during runtime. That is why the library is called I-See-Bytes. This variable, which is called a "**fluid**" (to distinguish it from ordinary C/C++ variables) holds bunch of bytes and the way it interprets (sees) those bytes at runtime changes its contents radically. Every variable type in the I-See-Bytes library, other than the basic C++ data types are of type ICBYTES. You never need to deal with other variable types such as:

```
Mat image(Y,X, CV_64F);
vector<Point2f> corners;
```

```
Mat screen(1080, 1920, CV_8UC4);
vector<KeyPoint> keypoints;
```

The other superiority of the I-See-Bytes library is its support for hardware. The OpenCV library has support for only USB and IP cameras. On the other hand, the I-See-Bytes library has support for the sound card, servo motor controllers, relay cards, etc. by using the only other class (object) in the library: ICDEVICE. You can use this object to establish internet sockets and access files as well. All of the functions in the I-See-Bytes library are smart; meaning that they can figure out what type of data structures there are in an ICBYTES type or ICDEVICE type fluid and operate accordingly. The user is not bothered by figuring out what type of input a function needs as an argument. In the OpenCV, if you send a wrong type of data structure to a function, it crashes without telling you what went

wrong. In the I-See-Bytes, the worst that can happen is that the function returns without doing nothing and returns an error value or a message explaining what went wrong. For example, in OpenCV, in order to convert a raw image format, you need to know the input type of your image. If you don't, then you can't use the following functions:

```
cvtColor(input,output,COLOR_RGB2ARGB); //from 24 bit color depth to 32 bit
cvtColor(input, output,COLOR_ARGB2GRAY);//from 32 bit color depth to 8 bit
```

To be able to use this function, you also need to know the following table:

COLOR_BGR2BGRA	COLOR_BGR2XYZ	COLOR_YUV2RGB_NV12
COLOR_RGB2RGBA	COLOR_BGR2YCrCb	COLOR_RGB2HLS_FULL
COLOR_BGRA2RGBA	COLOR_BGR2HSV	COLOR_YUV2RGBA_NV12
COLOR_BGRA2RGBA	COLOR_RGB2Lab	COLOR_YUV2BGR_I420
COLOR_GRAY2RGBA	COLOR_RGB2Luv	COLOR_BayerRGGB2RGBA
COLOR_BGR5552RGBA	COLOR_YUV2BGR	COLOR_BayerRGGB2RGB_EA

Unfortunately, OpenCV has a very bad documentation and good luck finding this table on the web. Finding an explanation for each of them??? On the other hand, in the I-See-Bytes library, you don't need to know what type of raw image you are sending. All you care about is the output. The library is smart enough to figure out the image type and employ the appropriate conversion mechanisms itself:

```
ToRGB32(input, output); //From any color depth to 32 bit color
ToRGB24(input, output); //From any color depth to 24 bit color
Luma(input, output);   // From any color depth to 8 bit grayscale color depth
```

To sum up, I-See-Bytes is a far superior library to OpenCV because of its Python like ease of use, tremendous GUI support and ability to interface with wide variety of hardware, enabling the user to write Robot Vision programs without the need to learn or include any other APIs or libraries for the hardware or the GUI interface.

Finally, the I-See-Bytes library uses the lowest level of Windows Kernel API, the WINAPI. That not only makes this library very fast, but also enables the compiled .exe programs to run on LINUX operating system (the system must have Intel compatible CPU) when used with the WINE (**W**ine **I**s **N**ot an **E**mulator) compatibility layer. At the moment, the I-See-Bytes library does not have as many different vision algorithms as the OpenCV, but given time, it will and it will contain the newest, fastest and most dependable of them.

Ibrahim Cem (Gem) BAYKAL

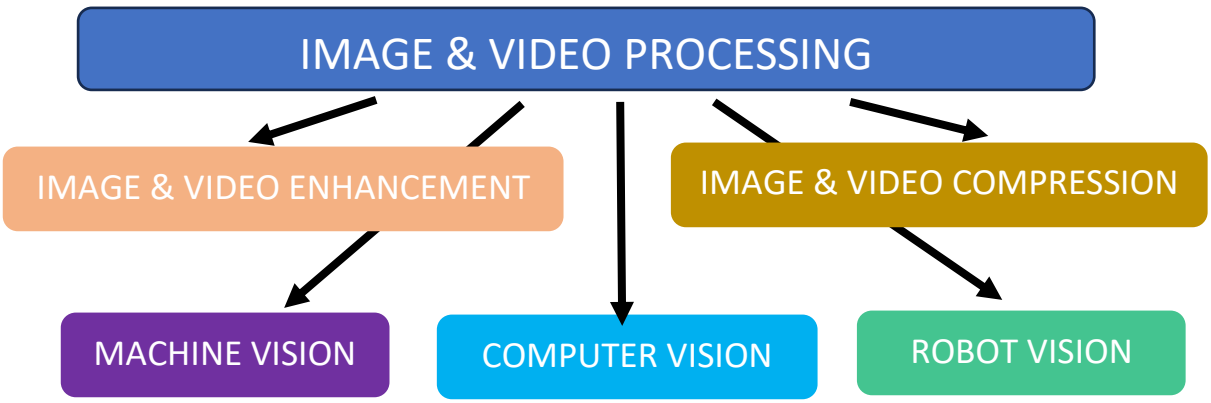
April 2025, Adana.

1. Terminology & Philosophy

Computer Vision (CV) is one of the most popular topics in Computer Science. However, there is a confusion about what it really means and what its difference is compared to the other similar fields such as the Machine Vision (MV), Image Processing and Robot Vision (RV).

Some people think that CV is the combination of Image Processing with Artificial Intelligence (AI) or Machine Learning (ML). Some people think that there is no difference between CV and MV. The main reason for this confusion is the CV researchers in academia who has never worked in MV or RV industry. Through their naivety, they think that because the same algorithms are used in these fields, they are the same. For MV experts who worked in the industry for years, distinction is clear. Unfortunately, the academia does not follow the industry. They coin their terminologies as they please and knowing that they are the real experts they expect the rest of the world to follow their lead. That is why you can see CV research articles in MV Journals.

First of all, Image and video processing is the field that generates tools for all the other related fields. Areas such as “Image Enhancement” and “Feature Extraction” are used by all the other fields. Since there is no restriction for Image Processing on the use of AI and ML, we can say that Image Processing is the name of the general field, and the rest of them are subfields of it. We may also include Feature Extraction and Pattern Recognition as subfields.



The main difference between the MV and the CV is that, basically, MV is CV with strict environmental and quality definitions. In Machine Vision, illumination conditions, the quality of the camera and even the objects that can come in front of the camera are strictly controlled. Machine vision is usually employed in industrial or commercial settings and used generally for purposes such as quality or process control. Machine vision systems have special lights that are specifically designed for a single task. The wavelength, intensity, dispersion and even the angle of the lighting is strictly controlled. On top of all these, their cameras are different too. They are called “Industrial Machine Vision Cameras” and not only have they very high quality, but also dozens of parameters that can be precisely altered by the user.

A researcher who has never used an industrial machine vision camera may not even get a proper image from an “Industrial” machine vision camera because

Figure 1.1. Three Industrial Machine Vision Cameras and a lens. From left to right: A CMOS line-scan Camera, its lens, a zoom controllable area-scan camera with integrated optics and a USB high-resolution auto-focus camera.

Figure 1.2. The back sides of the cameras shown in Figure 1.1.

of those parameters. Industrial Machine Vision cameras are so specialized that they are usually sold without lenses. The user must contact another vendor, an optics company, and purchase a specific lens that is most suitable for the application. He/she must know how to calculate even the lens parameters. On top of these, the user must know different camera technologies such as the CCD, CMOS and Time-Delay and Integration. He/she must decide whether to buy an area-scan or a line-scan camera. On the other hand, a cheap ordinary security camera is enough for a computer vision application. Figure1.1 shows three very different industrial MV cameras that are used for entirely different applications. For a seasoned MV expert, these are only a few of the types available among hundreds of cameras if not thousands.

These cameras not only have very different properties and parameters, but their connections are usually very different as well. While majority of the industrial MV cameras have USB or Gigabit ethernet connections (GigE, including 5 or 10 GigE), some cameras have so much data output that they have to be connected through different interfaces such as the Camera Link or CoaXPress (CXP). Their power consumption may be too high for the USB or ethernet cable to handle, therefore, they might require extra power connections. On top of that they might require synchronization signals generated by shaft encoders or pulsed illumination sources. Such signals are vital especially for Line-Scan Time Delay Integration (TDI) cameras.

The resolution and frame rate of the machine vision cameras are nonstandard. Vendors can produce cameras with arbitrary resolutions. They can be extremely large or very small for high frame rate cameras. Frame rates can

reach over 200fps. As a result of these, they require special expertise, otherwise, the image quality can be extremely poor.

Machine vision systems are usually employed in factories where a conveyor belt constantly spews out a single product. The objects that the camera sees can be strictly controlled. In a factory where bottles are produced, there is almost no chance of a bird flying in front of your camera. Because of such high-quality demands and strict environmental controls, MV systems have surpassed the human visual perception long ago. They can see better, faster and never get tired. On the other hand, majority of the Computer Vision systems are far from reaching the level of human visual abilities.

Figure 1.3 shows typical security cameras and a USB web cam. All of these cameras have a very simplistic interface (almost plug and play) and can be used for computer vision. Many of them doesn't even need an external power supply, the 5V through the USB or the voltage through the ethernet cable (if supports POE) is enough.

However, there is one crucial difference between these cameras and the machine vision ones that students must know: MV cameras have a fixed pixel gain while the ordinary ones have adaptive. When you first turn on an ordinary camera, the picture looks dark and it gradually gets brighter or when an ordinary camera is in the dark and you suddenly turn on the lights its picture gets ultra

Figure 1.3. Three security cameras and a webcam. From left to right: A Logitech USB webcam, a Wisenet, a Dahua and a HikVision outdoor security cameras.

bright, almost completely white and then slowly darkens and gets normal. That is because ordinary cameras adjust their pixel gains based on the ambient light in order to keep their dynamic range wide. On the other hand, MV cameras have fixed pixel gain (brightness) set by the user. Combined with precisely adjusted illumination conditions, this allows the user to employ simple pixel thresholding to detect colors. You cannot do that with an ordinary camera because its gain floats.

Some of the industrial machine vision cameras are so advanced that they can adjust the gain and bias of every individual pixel on their sensor. Combined with a color calibration process, their pixels will generate exactly the same pixel values when they see the same color. This means that, when you put a sheet of single-color paper in front of them, all of the pixel values in the image are the same number (say 197). Computer researchers who are unaware of that capability of such cameras will strongly object to using simple pixel thresholds out of their ignorance. Some of them are so fanatical about using a fixed threshold that they will accuse MV researchers of being unscientific. Using fixed thresholds are so common that some industrial MV cameras have that implemented inside them to reduce the data bandwidth. These cameras exclude the background (normal) pixels and send only the images of defective area by simply thresholding the image.

On the image processing side, MV researchers/engineers must have a better understanding of 2-D and statistical signal processing compared to CV ones. We can say that an MV engineer must know the mathematical basis of Image Processing better than the CV engineers. MV researchers are more like industrial R&D researchers rather than academical ones. They must be more hands on, and familiar with the hardware technologies. An industrial R&D engineer is more like a carpenter while an academician is more like a furniture designer. You may have graduated from the best furniture design school in Italy but that doesn't mean that you have hands on skills to build a dining table. These two skills are quite different.

1.1. Robot Vision

Having seen the difference between the Computer and Machine Vision, we can now ask what the difference of Robot Vision (RV) is. There are two main differences of this field: First of all, as the name implies, the vision is performed for a robot, meaning that the researcher or the programmer must command an

electromechanical, hydraulic, pneumatic of even organic actuator. In other words, the vision process must move a robot at the end of its completion. This requires some knowledge of hardware along with Control Theory and similar fields.

The second difference is that RV is almost exclusively performed on 3D data and results are in 3D too. A robot must always be aware of the depth dimension. Otherwise, it will crash into objects or break them. It may harm itself or people around it. This means that robot vision uses depth sensors other than cameras. Most common and popular of those sensors are LIDARs. They may also employ stereo cameras with integrated depth calculators or similar devices, along with other, simpler sensors such as ultrasonic, pressure or light reflection sensors.

WARNING: OPERATING ROBOTS MAY BE EXTREMELY DANGEROUS!

- A powerful robot must always be operated inside a metal cage where humans are not allowed to get inside. Otherwise, it may hit humans and injure them.
- A mobile robot must have fail-safe sensors that protects them from crashing into objects. Otherwise, it may crash into humans/pets and injure them.
- Remove all living beings, containers of flammable or corrosive liquids and fragile objects around the path of the mobile robots.
- Mobile robots contain explosive batteries like lithium-ion. These batteries will burst into flames spontaneously or explode when punctured. You must take necessary precautions.

THE READER BEARS ALL RESPONSIBILITY WHILE CONDUCTING ANY EXPERIMENTS USING THE PROGRAMS IN THIS BOOK OR THE I-SEE-BYTES LIBRARY.

1.2. The Philosophy of Engineering

There are thousands of engineering faculties at the universities all around the world. These faculties focus on educating their students and prepare them with skills so that they become desirable employees for engineering companies. Some of their students attend those universities because they enjoy working in those fields; they wonder how machines work, they want to know how the universe works. But the majority of them attend because they want to be employed, have regular salaries and live a comfortable life. In other words, they want to make money.

Aware of this motivation, especially those universities with high tuition fees, teach courses with an emphasis on how these courses will propel students to their desired jobs, how technologically cutting edge their education is. They do not stop for a second to consider what the philosophy behind engineering is. You would be surprised how few of the professors know what the reason behind why we spend so much time, money and energy on research and education. If you ask a professor why we do science, he or she will most probably say "to enlighten the human kind."

Enlightening the human race is definitely one of the reasons and a noble one. But it is nowhere near the main reason. The main reason we do science and engineering is **to enhance the quality of human life**. Consider the person who earns minimum wage. Governments all around the world collect taxes from these people and use it to fund scientific research and universities. Every penny, cent or kuruş of tax being collected from those people is less money that those people can spend on their kids**. In other words, the money that could have been used to feed the kids of the taxpayers is being used for the research and education at the universities**.

The lesson here is that science must be conducted to solve the humanities', or the taxpayers' problems. The duty to enlighten the public is a far less important job of a scientist. A professor must know where his/her research money is coming from and feel the responsibility.

Now that we know the philosophy behind why we conduct scientific research, let's focus on engineering. Engineering is the application of scientific knowledge in order to design things that enhance, simplify or speed up human

life. This part of the definition is common knowledge and well-known by both students and professors. What is missing in this definition, that is not being thought at most of the universities is that **engineering must be performed using minimum resources**. In other words, engineering is the application of our scientific knowledge to enhance human life by using minimum resources.

We are all aware of the devastating effects of pollution. The ecosystem of this planet cannot recover from the huge amount of toxic byproducts that our industry produces. Not only the global warming caused by excessive carbon dioxide, but also other chemicals are destroying natural life at a pace unseen in human history. On top of that, our resources are diminishing at an alarming rate. Prices of oil, copper, zinc and rare metals like antimony are rising because the reserves are decreasing while demand increases. Increasing human population and environmental decay has put stress on the amount of fresh water and agricultural production. That is why constructing designs that use less resources are important for an engineer more than ever before.

However, a less known fact among engineers is that using less resources is important for your own company and country. Imagine three car producers from Germany (yellow), Türkiye (turquoise) and China (red). Let's assume that the first two (German and Turkish) cars are being sold for 10K dollars. But the German car has superior quality to the Turkish car. It consumes less gas per kilometer (or mile) and it doesn't require parts change as often. It produces more power for the same size (liters) engine and it is safer and more comfortable. Can the turquoise car be sold? Would customers prefer the Turkish made car over the German one? The answer is a simple big NO. Customers would try to get maximum for their money. Therefore, they would buy the superior German car because they cost the same. What would happen to the Turkish manufacturer? They spent a lot of money to build a factory, they paid to the workers and purchased raw materials for the car. They need cash to turn the wheels. Obviously if the company has no income, it would go bankrupt.

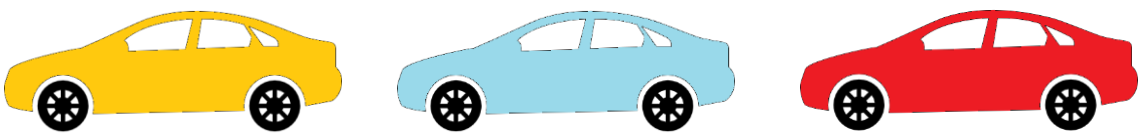


Figure 1.4. Three cars manufactured by three different car makers from Germany, Türkiye and China.

But wait. There is another solution. The Turkish government can (or be forced to) reduce the value of the Turkish Lira. If the Lira is devalued to %85 of what it was, then the Turkish car would cost 8500 dollars. Because it is cheaper, some customers would prefer the Turkish car over the German one and the Turkish car manufacturer can stay alive. Workers will get paid (less in dollars but same in Liras) and their families will not become (very) poor.

Let's consider another scenario. Let's say that the Turkish and the Chinese cars have almost the same quality but the Chinese car is being sold for 8000 dollars because it costs less. Would the customers prefer Turkish made car? Again, the answer is a big no (unless the Turkish car is called iPhone). Why would you pay more for the same product? You may ask how the Chinese can produce the same quality car for much less cost? They may be using cheap labor, or they may be using an efficient, automated production infostructure. In other words, they might be using **robots** to lower their costs.

Robots are increasingly being used in manufacturing, especially for the production of motor vehicles. Toyota is one of the leading companies in using robots, helping them become the worlds' biggest car maker. Robots do not get tired, sick or depressed. They don't strike when they aren't paid enough. They work 24 hours a day and 353 days (once a month maintenance day) a year.

1.3. The Design of the I-See-Bytes

I-See-Bytes is a Robot Vision (RV) library designed to lower the cost of not only research and development but also teaching. In order to be efficient, a Robot Vision library needs to be very fast because it processes huge amount of data in real-time. We all know that the fastest high-level programming language is C++. Unfortunately, C++ is difficult to use. That is why majority of the universities started teaching Python or Matlab as the introduction to programming courses in the engineering departments. As mentioned in the book titled "I-See-Bytes a Simplified C++ Library;" despite being a C++ library, it has many of the comforts of Python and Matlab. The main class of this library, which is called a fluid, can be used to hold any variable type and different containers / data structures such as matrixes (multidimensional C++ arrays) and lists (2D C++ STL vectors).

Unlike the OpenCV, there is a single data type (called ICBYTES) and all of the functions receive or return this same data type. This not only decreases the time spent for writing code, but also makes it easier for the instructors to teach this library to the new students. OpenCV has a Python version but because Python runs on virtual machine, a code written in Python works 10 times slower than the same code written in C++. Robot vision applications require huge amount of processing power and if using Python makes your product 10 times slower than your competitors, your company will lose its competitive edge from the start.

You may say that you will use hardware that is 10 times faster. Then your costs will go up because you will need to use 10 CPUs in your motherboard while your competitors use only one. Things get even worse if you are designing a mobile robot. A mobile robot runs on batteries and if you use 10 CPUs on your design then your batteries will run out 10 times faster. Who would want to buy a robot that is slow, expensive and runs out of battery faster than other robots?

1.3.1. High Quality GUI

Another important superiority of the I-See-Bytes library is its simplified interface to the Windows API. The Windows API (used to be called the Win32 API) is the programming interface used to create and use graphical user interface (GUI) elements such as buttons, edit windows and image frames. Other libraries such as the OpenCV has little or no support for GUI making it impossible to create professional looking applications. On the other hand, because the I-See-Bytes library is designed based upon the Windows API, you can not only create professional looking GUI interfaces, but also use the Windows API (if you know how) to create your own custom interface without any compatibility issues.

Using the Windows API at the heart of the I-See-Bytes library has two other important advantages. First of all, it is very fast. The Windows API is the lowest level programming interface of the Windows operating system with most of its calls directly tied to the operating system kernel. That makes this interface considerably faster than other interfaces or GUI libraries such as the QT or GTK+. If you want to display several HD videos on the screen or 3D animations, such advantages will make your application faster than your competitors.

The third advantage of using the pure Windows API is cross platform compatibility. As we all know, when the Java language came out, they promised that the Java bytecode would be executable on any platform, Windows, Linux or MacOS. As time passed, Java failed to deliver on that promise. Fortunately, the Windows API has been around as long as the Windows has been and people had

a long time to reverse engineer it. Consequently, a famous (mostly for Linux users) compatibility layer has been written called the **WINE** (**W**ine **I**s **N**ot and **E**mulator). WINE is an opensource compatibility layer that converts Windows system calls (Windows API calls) to the host operating system. As a result, if you are using Intel family CPU and Linux or MacOS operating systems, you can run Windows applications once you install WINE on your computer. The executables you create using the I-See-Bytes library run smoothly on both Linux and MacOS (with intel CPU) and at the same speed. That is something Java cannot do.

1.3.2. Direct Hardware Access

One of the most important advantages of using the I-See-Bytes library is its ability to control the robot hardware directly. A single class called "**ICDEVICE**" (pronounced I-See-Device) can be used to access not only cameras but also relay cards, LIDARs, servo controllers, microphones and even network devices.

As we will see in the following chapters, the I-See-Bytes library can control relay cards that are used for turning on and off heavy-duty devices. You can control even a refrigerator using a relay card. If you connect the motors of a mobile robot to a relay card, then you can control the movements of that robot. Servo motors can also be controlled directly, making robot arms very easy to use. Combined with vision algorithms, ability to receive data from cameras and sensors such as LIDARs, the I-See-Bytes library allows you to develop robot vision applications without the need to learn or integrate any other interface.

1.3.3. Fastest, Reliable and Newest Vision Algorithms

Every year thousands of new vision algorithms are published in scientific journals. Unfortunately, we seasoned vision scientist know that, even the algorithms published in the highest ranked journals do not work as reliably as advertised in their articles. If an algorithm works on 70 images out of 100, its inventors chose to publish those 70 images and only a few of the 30 that doesn't work, out of the pressure to publish their articles. On top of that, article publishing business is not a moral one, that favor western universities, especially the ones from USA and Britain as these countries make a lot of money from foreign students and those students want to attend a high-ranking research university.

The other problem is that the academicians are not worried about getting their company bankrupt because they don't work at a company. They are focused on creating algorithms that work more successfully rather than

increasing the processing efficiency of their algorithm. As a result, you see algorithms published in respectable journals that improve the accuracy of an existing algorithm only %10 while increasing the processing power consumption %300. As an engineer, if you design a system that works only %10 more accurately than your competitor but increase hardware costs %300 percent, your boss would fire you. Why? Because there are many different algorithms with completely different approaches that do the same thing. Instead of using an algorithm that increases the cost %300, it is much smarter to use two completely different, parallel algorithms as shown in the figure below and fuse both algorithms at almost half the cost.

Because those two algorithms have completely different approaches, one of them will most probably correctly process the image that the other did not. Therefore, if you have very different two algorithms with accuracies of %80 and %65, a smart decision-making algorithm based on their confidences may combine the results producing an overall accuracy of %90. The combined processing power needed would be %100 + %70 (plus some for the decision-making algorithm) equaling %180. That is why, in the I-See-Bytes library, only (mostly) the fastest and most accurate or in other words, the most efficient vision algorithms are included. **Crappy algorithms like the Hough Transform** will never be included in the I-See-Bytes library.

As we mentioned before, every year thousands of algorithms are being published. Unfortunately, it takes a long time for these algorithms to be included in opensource libraries such as the OpenCV because people work in those projects as volunteers and as a result, the time they spent on them is limited. The I-See-Bytes library is supported by both the private sector and the universities and hopefully that will help this library to include newest algorithms faster than the opensource alternatives.

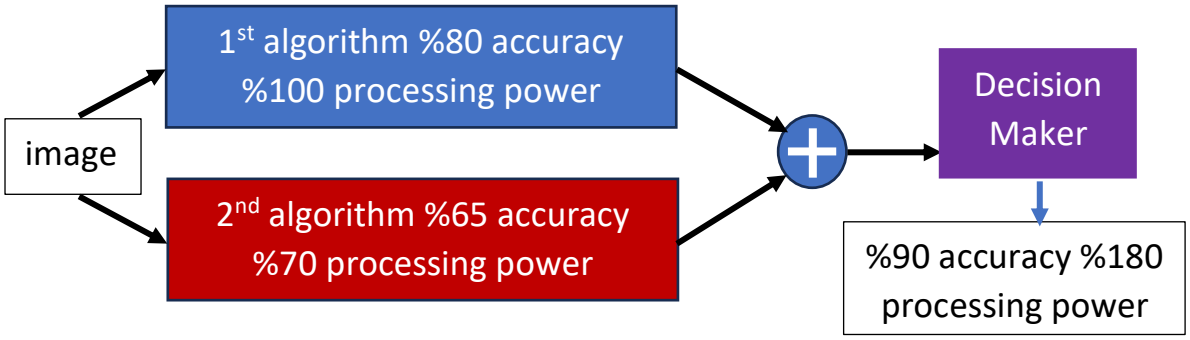


Figure 1.5. Two very different vision approaches can be fused together to result in a more successful vision algorithm.

2. Digital Signal Processing

Any digital image is considered a two-dimensional digital signal. Therefore, students need to understand the concepts and the techniques of Digital Signal Processing (DSP) in order to become good engineers. All signals are analog in nature. For example, when you speak to a microphone, the changing pressure caused by the vibration of air creates electric signals in the range of millivolts. If we plot this signal's magnitude with respect to time, we get graphs as shown in Figure 2.1. This signal is continuous in both value and time; meaning that between 345th and 346th milliseconds, there are infinite number of values.

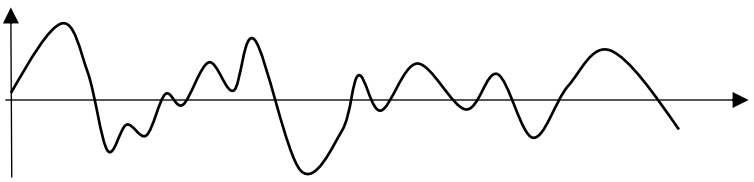


Figure 2.1. A continuous sound signal coming from a microphone.

What if we want to process this signal inside a computer? We know that everything inside a computer consists of numbers. Even the letter "A" is actually number 65 that is being interpreted as a character. This means that we need to convert the signal to numbers before processing it. The way we can do that is by measuring the voltage of the above signal with a voltmeter. A voltmeter tells us what the voltage of an energy source is. For example, if we connect a voltmeter to an everyday AA type battery, we will measure around 1500 millivolts.

The problem with a microphone signal is that it changes very quickly. Humans can hear sound frequencies between 20 Hz and 20 KHz. A 20 KHz signal alters its value **twenty thousand times a second**. Therefore, what we need is a voltmeter that can read values at twenty thousand times per second and record them. Fortunately, your computer is equipped with such a voltmeter. It is called

the sound card. When you talk into a microphone connected to your computer's sound card, the sound card measures those millivolt range signals and sends them to your computer's RAM. From there, you can either play those sounds or save them on your hard drive. You can change the base and treble of the sound. The base is low frequency signals such as drums, and treble is high frequency signals such as the sound of violin. So how does the music player on your computer increases base or treble? That is done by applying DSP techniques.

2.1. Removing Signals From Each Other

When we work with sounds, we actually work with sine signals. That is because any sound can be constructed as a sum of different sine signals. In this experiment we will write an application that generates two sine signals with different frequencies, then add those signals, and then try to separate them again. The first signal we generate is a sine signal with a period of 12 as shown below.

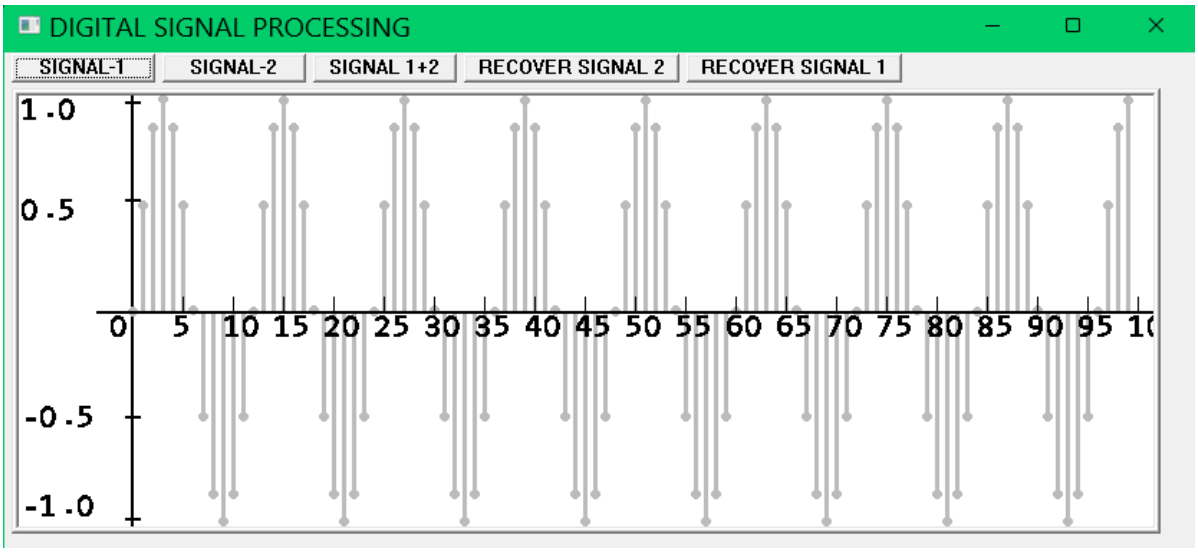


Figure 2.2. Screenshot of the program displaying a sine signal with period 12 after pressing the button "SIGNAL-1."

The code that generates this signal is shown below:

```
#include"icb_gui.h"
#include<math.h>
int FRM;
void ICGUI_Create()
{
      ICG_MWSize(850, 400);
      ICG_MWTitle("DIGITAL SIGNAL PROCESSING");
}
```

```
void Signal1()
{
	static ICBYTES signal1,graph;
	CreateMatrix(signal1, 100, 1, ICB_DOUBLE);
	for (int x = 0; x < 101; x++)
		signal1.D(x + 1) = sin(3.1415 * (double)x / (6.0));
	BarChart(graph, signal1, 300, 2, 0xffffff, 0xbbbbbb);
	DisplayImage(FRM, graph);
}
```

The above function, "Signal1()," generates the sine function shown in Figure 2.2 and displays it. The second function, Signal2(), is called by the second button with the same name and draws another sine signal:

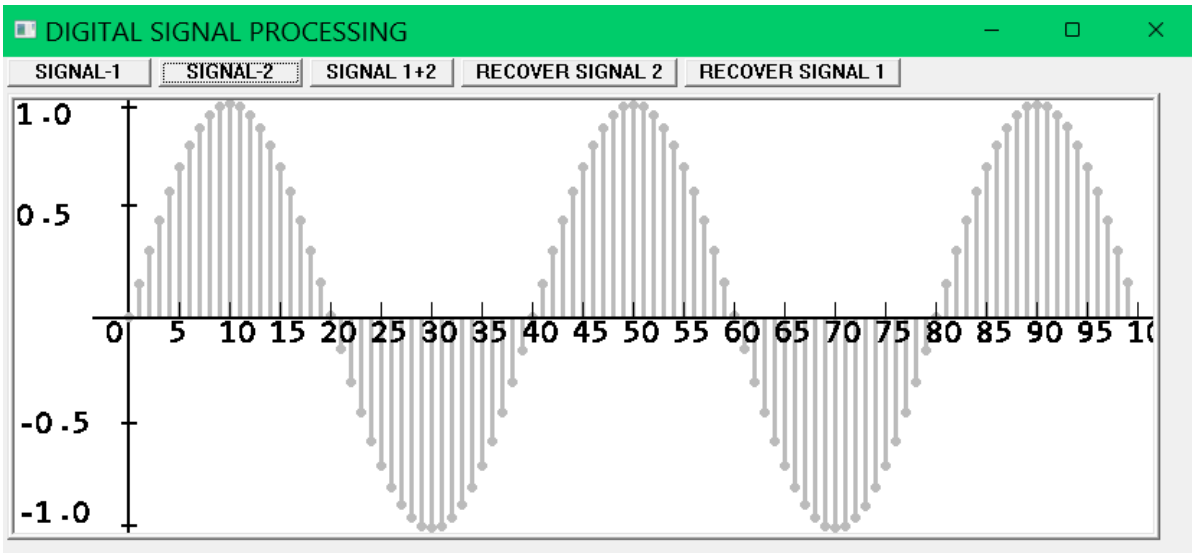


Figure 2.3. Screenshot of the program displaying a sine signal with a period of 40 after pressing the button "SIGNAL-2."

The code that generates and displays the above graph is:

```
void Signal2()
{
	static ICBYTES signal2, graph;
	CreateMatrix(signal2, 100, 1, ICB_DOUBLE);
	for (int x = 0; x < 101; x++)
		signal2.D(x + 1) = sin(3.1415 * (double)x / (20.0));
	BarChart(graph, signal2, 300, 2, 0xffffff, 0xbbbbbb);
	DisplayImage(FRM, graph);
}
```

If we multiply the second one with 1.1 (increase its magnitude %10) and add these two signals, the resultant signal is shown in Figure 2.4.

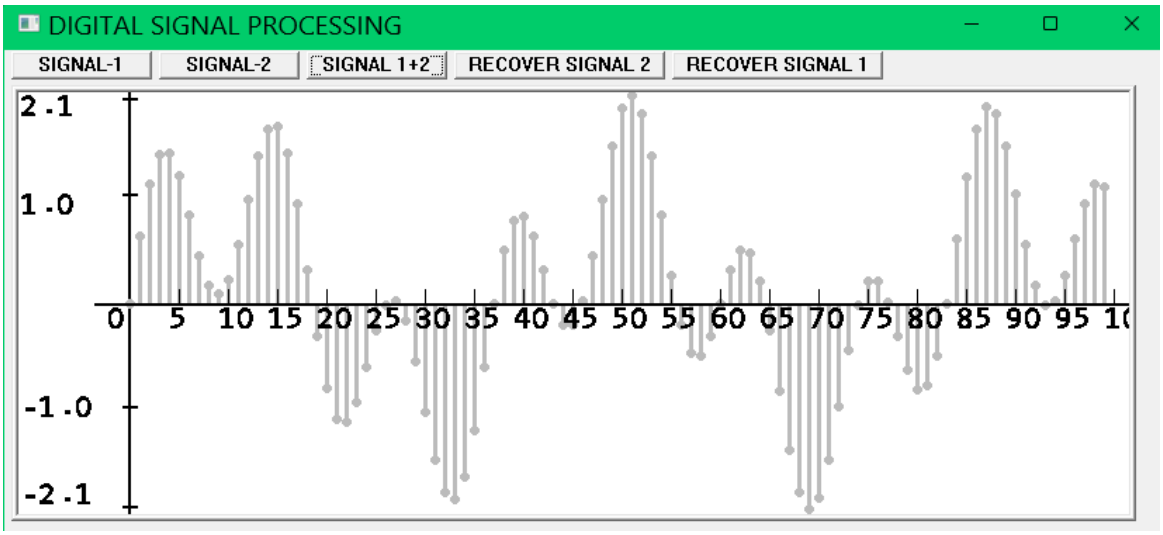


Figure 2.4. The screenshot of the program displaying sine signals with period of 12 plus a period of 40, after pressing the button "SIGNAL 1+2."

The function that is called when the button labelled "SIGNAL 1+2" is shown below:

```
void Signal1_2()
{
 static ICBYTES data;
 static ICBYTES graph;
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 for (int x = 0; x < 101; x++)
   data.D(x+1)=sin(3.1415*(double)x/(6.0))+1.1*sin(3.1415*(double)x/(20.0));
 BarChart(graph, data, 300,2,0xffffff,0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

As you can see from the previous graphs, Fig 2.4 resembles neither the sine signal with period 12 nor the one with 40 samples. So how do we separate these two components if we were given only the combined signal?

EXERCISE: Before reading further, students should try to come up with a way (program) to separate these two signals.

HINT: When summed over an exact entire period (or integer multiples of it), the result is always zero. Otherwise, it is never zero.

2.2. FILTERING

Filtering is essentially shifting one short signal over a longer one and multiplying and adding sample values. The short signal is called “The Filter” and it can be something like this:

filter={ 0.1, 0.2, 0.3,0.2,0.1}

signal={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

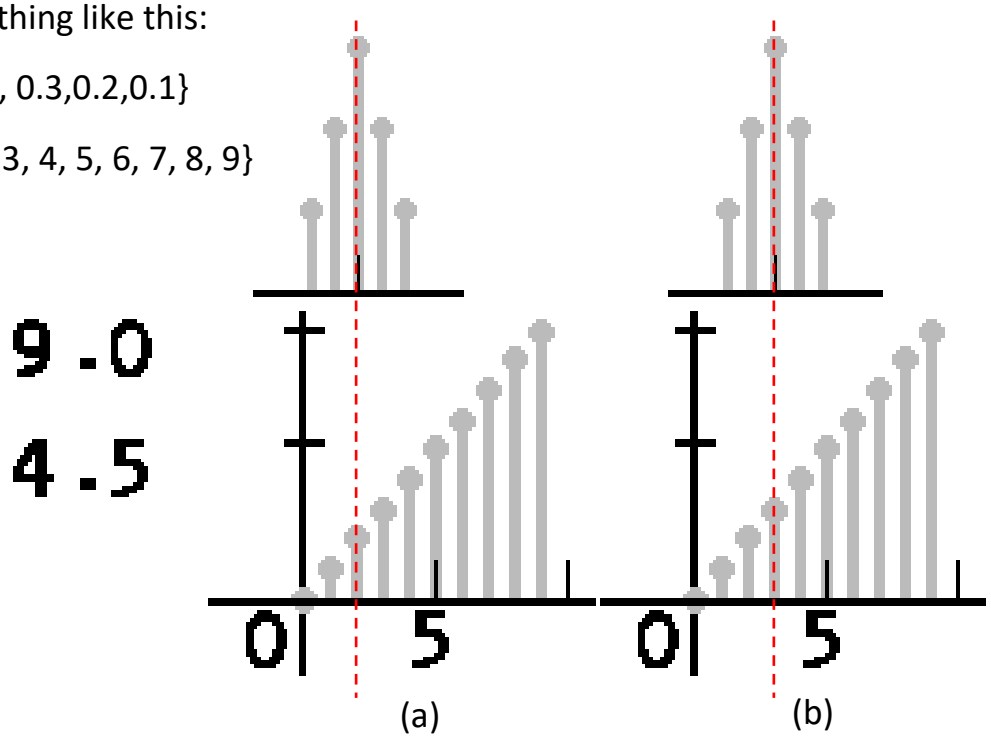


Figure 2.5. The process of filtering is demonstrated using two simple signals. The filter is shifted towards the right and coinciding samples are multiplied and summed to calculate each output sample.

Let’s assume that we wish to calculate the output of the filter at t=2. (Figure 2.5(a)) At this instance, the signal value is 2. We bring the middle of the filter, which is the third element (0.3) on top of t=2 and we start multiplying and adding the elements of each signal that coincide upon each other:

$$F(2)=0\times0.1 + 1\times0.2 + 2\times0.3 + 3\times0.2 + 4\times0.1$$

$$F(2)=1.8$$

Then, to calculate the output at t=3, we shift the filter signal one step to the right making the t=2 (0.3) sample of the filter with t=3 (3) sample of the signal. This time the output at t=3 is:

$$F(3)=1\times0.1 + 2\times0.2 + 3\times0.3 + 4\times0.2 + 5\times0.1$$

$$F(3)=2.7$$

2.3. Filtering the Second Signal

Since we know that the sum of a sine signal over its entire period is zero, we can use this knowledge to design a filter. The first signal has a period of 12, meaning that, if we average every 12 samples in the combined signal, the first signal's values will cancel each other out and all that remains will be the second signal. Below is the function that first generates the combined signal and then extracts the second one out of it:

```
void Recover2()
{
 static ICBYTES data, out;
 static ICBYTES graph;
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 CreateMatrix(out, 100, 1, ICB_DOUBLE);
 out = 0;
 for (int x = 0; x < 101; x++)
  data.D(x+1)=sin(3.1415*(double)x/(6.0))+1.1*sin(3.1415 * (double)x/(20.0));
  for (int x = 6; x < 95; x++)
   for (int z = -5; z <= 6; z++)
      out.D(x) += data.D(x - z) / 12.0;
 BarChart(graph, out, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

The output of this function is stored in the "out" vector and its graph is shown below in Figure 2.6. Since the middle of the moving average filter is at the 6th sample, and the entire signals must be on top of each other, the first signal that can be calculated is the 6th signal of the output which is at t=5. The slight distortion at the beginning is caused by the fact that 12 is not an odd number. If the filter's length is not odd, then such distortions might occur at either ends of the output signal. Also, you may notice that the location of the peak of the signals may shift as well.

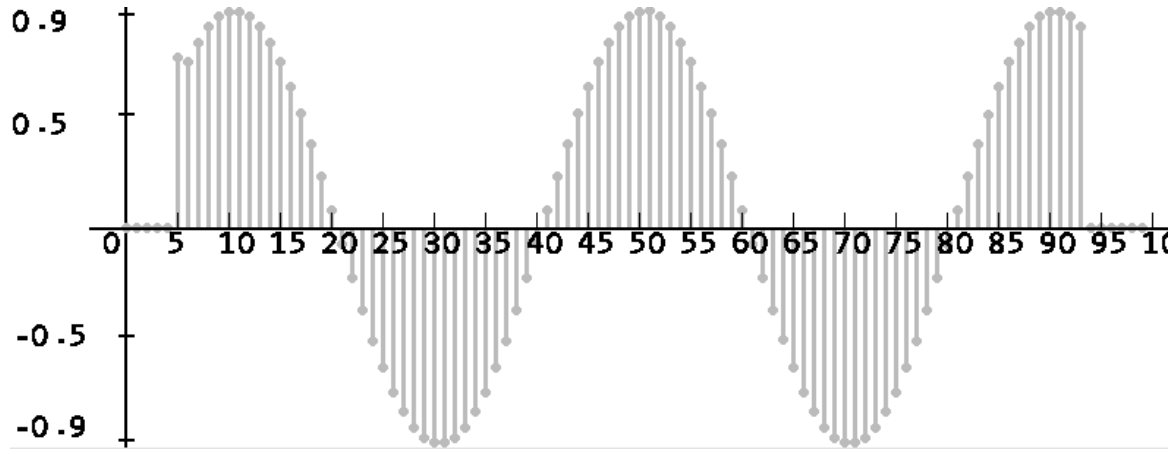


Figure 2.6. The recovered signal from the combination of two sine signals that was shown in Figure 2.4.

So how do we recover the first signal? We can do the same thing we did for signal-1 and create a longer moving average filter to suppress signal-2. Unfortunately, that method might not work well because the period of the second signal, 40, is slightly close to the quadruple the period of the first signal, 12. Another method we can apply is to boost signal-1. For this, we use a filter as shown below:

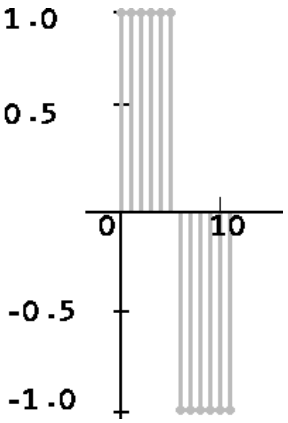


Figure 2.7. The high-pass filter (shifted) that boosts signal-1 and suppresses signal-2.

The code for this type of filtering is:

```
void Recover1()
{
 static ICBYTES data, out;
 static ICBYTES graph;
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 CreateMatrix(out, 100, 1, ICB_DOUBLE);
 out = 0;
 for (int x = 0; x < 101; x++)data.D(x + 1) = sin(3.1415 * (double)x /
(6.0)) + 1.1*sin(3.1415 * (double)x / (20.0));
 for (int x = 6; x < 95; x++)
 {
      for (int z = -5; z < 1; z++)
           out.D(x) += data.D(x - z);
      for (int z = 1; z < 7; z++)
           out.D(x) -= data.D(x + z);
 }
 BarChart(graph, out, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

When we apply this filter to the combined signal, the output of the process is shown below in Figure 2.8.

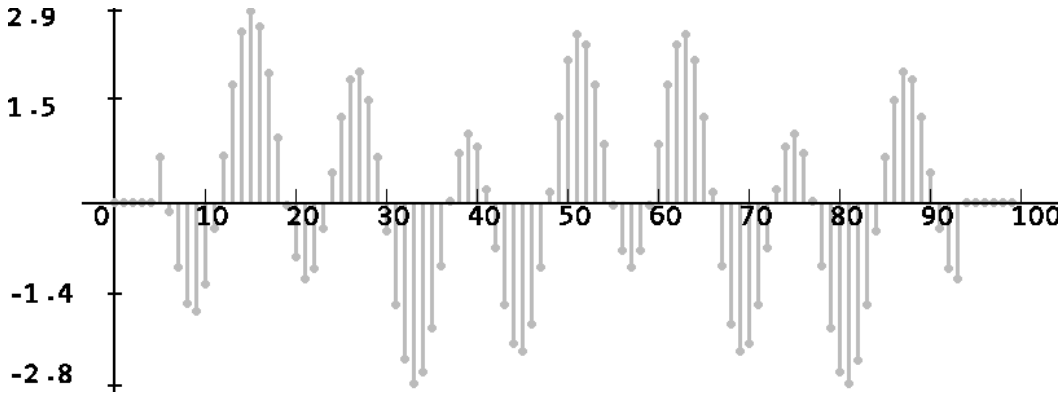


Figure 2.8. Output of the high-pass filter on the combined signal.

The rest of the code is shown below:

```
void ICGUI_main()
{
      ICG_Button(5, 1, 100, 20, "SIGNAL-1", Signal1);
      ICG_Button(110, 1, 100, 20, "SIGNAL-2", Signal2);
      ICG_Button(215, 1, 100, 20, "SIGNAL 1+2", Signal1_2);
      ICG_Button(320, 1, 150, 20, "RECOVER SIGNAL 2", Recover2);
      ICG_Button(475, 1, 150, 20, "RECOVER SIGNAL 1", Recover1);
      FRM = ICG_FrameMedium(5, 25, 160, 110);
}
```

2.4. What Does That All Mean?

In the previous sections we learned about one dimensional digital signals. We created two sine signals with different frequencies summed them up together and then try to separate that sum into its components by applying filters. What does that have anything to do with images? Images are digital signals too. The main difference is that it is a two-dimensional signal. One other less important difference is that image signal samples, which are called pixels, are usually positive integers. Image pixel values usually never go below zero. In C/C++, “unsigned char” or “byte” variable type is used for holding image signals. An unsigned char is an 8-bit unsigned integer that can hold values between 0 and 255. Therefore, if we modify our code to create a sine wave that encompasses this entire range, then we get the code below.

```
void Signal1()
{
 static ICBYTES signal1,graph;
 CreateMatrix(signal1, 100, 1, ICB_UCHAR);
 for (int x = 0; x < 101; x++)
  signal1.B(x + 1) = 128.0*(sin(3.1415 * (double)x / (6.0))+1.0);
 BarChart(graph, signal1, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

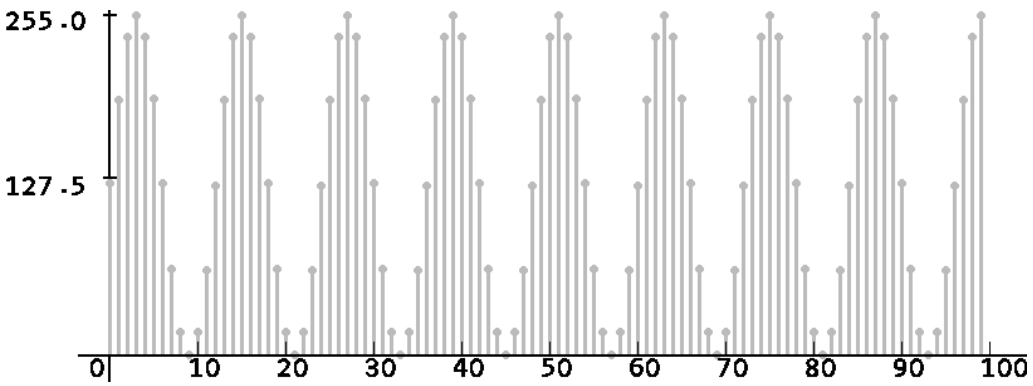
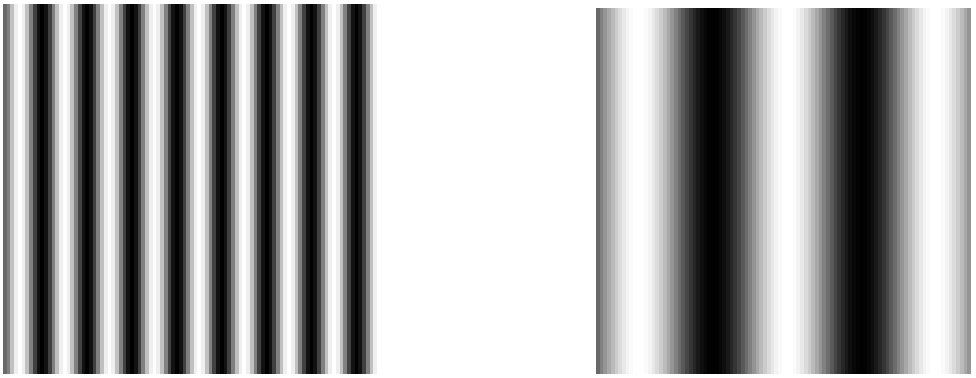


Figure 2.9. The graph drawn by the code in the previous page.

Now, what will happen if we use this signal as pixels? All we need to do is copy this signal as rows of a matrix and display that matrix as an image. This signal consists of 100 samples. If we replicate this signal 100 times for each row of a matrix then we will get a 100×100 element matrix which is a grayscale image of same size. Let's modify our code:

```
void Signal1()
{
 static ICBYTES signal1,graph;
 CreateMatrix(signal1, 100, 100, ICB_UCHAR);
 for (int y = 1; y < 101; y++)
 for (int x = 0; x < 101; x++)
  signal1.B(x + 1,y) = 128.0*(sin(3.1415 * (double)x / (6.0))+1.0);
 //BarChart(graph, signal1, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, signal1);
}
```

For a sine wave with a period of 12 pixels, we would get the image shown in Figure 2.10.(a) and for 40-pixel period (signal-2), we would get the one on the right, Figure 2.10.(b).



(a) (b)

Figure 2.10. An image whose horizontal pixels are defined by a sine wave with period 12 (a). Image produced same way with a sine wave of period 40 (b).

When we look at the images shown in Figure 2.10, we see that the sine wave with the shorter period (12) creates more rapidly changing gray tones (a). A sine wave with a smaller period means a signal with higher frequency. A higher frequency in an image signal means more rapidly changing contrast levels.

Now let's remember what the moving average filtered did to our combined signal1+signal2. It suppressed the high frequency signal1 and it had less effect on the signal2. Why? Because when we average the neighboring signal values, the negative ones cancel the positives and the signal cancels itself out. A high frequency sine wave has positive and negative values closer to each other but a low frequency signal has them so far apart that the positive ones are too far away to be canceled out by the negative ones and vice versa. On the other hand, a high pass filter subtracts the negative ones from the positive ones creating even higher values (1 - (-1) = 2), boosting the high frequency signals. Because the low frequency signals have similar values around a neighborhood, this time they start canceling each other and the low frequency signal gets suppressed.

2.4.1 Notation

The moving average filter we used in the previous section can be simply expressed as a function as shown below:

$$y[n] = \frac{1}{12}\sum_{i=-5}^{6} x[n+i] \qquad (2.1)$$

Since we are dealing with discrete signals, we used the ∑ sign to express that samples are going be summed. If this were a continuous analog signal, we would have used the following notation:

$$y(t) = \frac{1}{12}\int_{k=-5}^{6} x(t+k)dx \qquad (2.2)$$

This tells us that a moving average filter is an integrator. As a matter of fact, all (pure) low-pass filters are integrators and high-pass filters are derivative operators or difference operators for the discrete case. Based on this knowledge, we can design the simplest low pass and high pass filters:

y[n]= (x[n]+x[n+1])/2 is the simplest low-pass and,

y[n]= x[n]-x[n+1] is the simplest high-pass filter.

Note that we prefer to keep the sum of the coefficients 1 for of a low-pass filter and 0 for a high-pass.

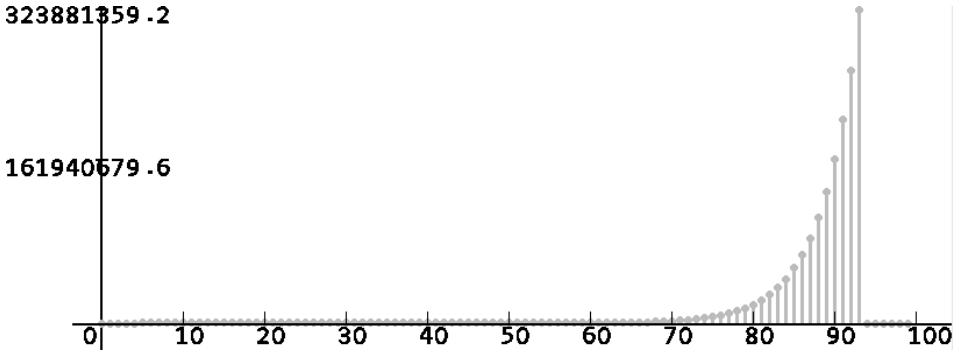
The general form of a Finite Impulse Response (FIR) Filter is:

$$y[n] = \sum_{i=0}^{m} a_n x[n-i]$$

This is a finite response filter meaning that if the input x[n] vanishes, or in other words becomes zero, then, eventually y[n] becomes zero as well. Opposite to this is an Infinite Impulse Response (IIR) filter:

$$y[n] = \sum_{j=1}^{p} b_n y[n-j] + \sum_{i=0}^{m} a_n x[n-i]$$

As you can see, this filter uses the previous outputs of the filter. That is why it is also called a recursive filter. In practice, just like an FIR filter, when the input vanishes, we would want the output of an IIR filter get so small that it can be practically considered zero. IIR filters are more difficult to design and they can easily become unstable, meaning that their output can fluctuate forever or approach infinity**. If you are not an expert, avoid designing them.** For example, do the following tiny change to the Recover2() function (highlighted with red color) and get the output shown below. Note that the if the input signal were in volts, the output would have reached almost 324 billion volts!




```
void Recover2()
{
 static ICBYTES data, out;
 static ICBYTES graph
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 CreateMatrix(out, 100, 1, ICB_DOUBLE);
 out = 0;
 for (int x = 0; x < 101; x++)
  data.D(x+1)=sin(3.1415*(double)x/(6.0))+1.1*sin(3.1415 * (double)x/(20.0));
 for (int x = 6; x < 95; x++)
  for (int z = -5; z <= 6; z++)
   out.D(x) += out.D(x-z)/3.0+data.D(x - z) / 12.0;
 BarChart(graph, out, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

2.5. Filters: What are They Good for?

In Figure 2.12, we see the famous mandrill image that has been used countless times for image processing demonstrations. We first convert the RGB (**R**ed, **G**reen, **B**lue) color image to luminance image. This image has 8-bit unsigned bytes to indicate the brightness of each pixel. Then we extract the first 200 pixel values from the 100th line (row) of this image and put them in a vector called "signal". If we display that vector as a signal we get the waveform shown at the bottom of Figure 2.12.

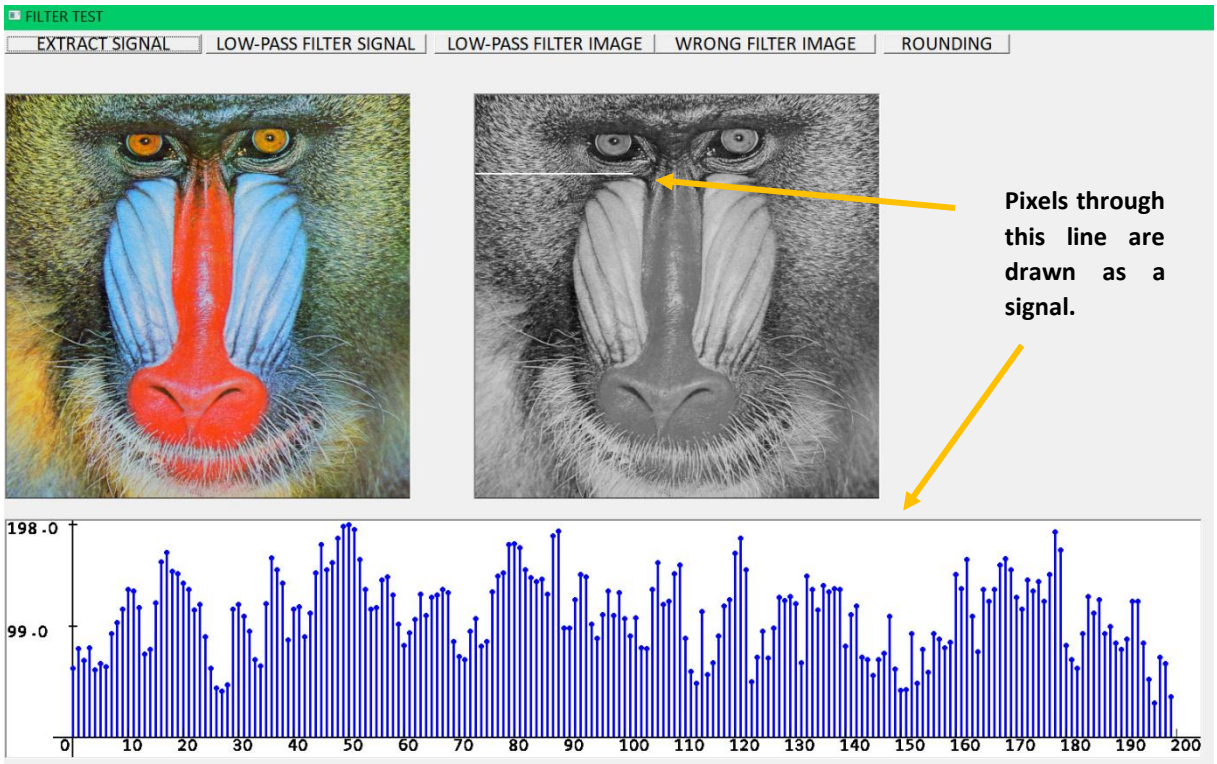


Figure 2.12. The screenshot of the program used for demonstrating low-pass filtering effects.

Below is the first part of the code for this program:

```
#include"icb_gui.h"

int MLE, SLE, FRM1, FRM2,FRM3;
ICBYTES signal;
void Extract();
void LowPassSignal();
void LowPassImage();
void WrongFilter();
void Rounding();
```

```
void ICGUI_Create()
{
      ICG_MWTitle("FILTER TEST");
      ICG_MWSize(1800, 1000);
}

void ICGUI_main()
{
  ICG_SetFont(30, 0, "Calibri");
  ICG_Button(5, 5, 250, 25, "EXTRACT SIGNAL", Extract);
  ICG_Button(260, 5, 280, 25, "LOW-PASS FILTER SIGNAL", LowPassSignal);
  ICG_Button(550, 5, 280, 25, "LOW-PASS FILTER IMAGE", LowPassImage);
  ICG_Button(830, 5, 280, 25, "WRONG FILTER IMAGE", WrongFilter);
  ICG_Button(1120, 5, 160, 25, "ROUNDING", Rounding);
  FRM1 = ICG_FrameSunken(5, 80, 512, 512);
  FRM2 = ICG_FrameSunken(600, 80, 300, 300);
  FRM3 = ICG_FrameSunken(5, 620, 1700, 300);
}
```

Now let's see the "Extract()" function that extracts the 200 pixel values from the 100th line of the grayscale image:

```
void Extract()
{
      ICBYTES RGB, grey, graph;
      ReadImage("mandrill.bmp", RGB);
      Luma(RGB, grey);
      DisplayImage(FRM1, RGB);
      CreateMatrix(signal, 200, ICB_UCHAR);
      for (int x = 1; x <= 200; x++)
      {
            signal.B_[1][x] = grey.B_[100][x];
            grey.B_[100][x] = 255;
            grey.B_[101][x] = 255;
      }
      DisplayImage(FRM2, grey);
      BarChart(graph, signal, 300, 2, 0xffffff, 0xff);
      DisplayImage(FRM3, graph);
}
```

This function creates three fluids: RGB, grey, and graph. The RGB matrix contains the color mandrill image. Note that the "mandrill.bmp" must be inside the Visual Studio project directory. After the image is load, it is converted to grayscale image using the Luma(.) function and copied into the "grey" matrix. Then a vector of size 200 is created inside the global "signal" fluid. The following line inside the for loop:

```
signal.B_[1][x] = grey.B_[100][x]; //using fast access
```

(or you could use: `signal.B(x)=grey.B(x,100);` for slow but secure access.)

tells us that the pixel values of the 100th line in the grayscale image is being copied into the "signal" vector. The next two lines are making the pixels of the 100th and 101th lines (in order to make the white line double thickness so that it can be seen better) pure white so that we can see where we are extracting the signal. Then the signal is drawn on the "graph" matrix using the BarChart(.) function.

This bar chart graph tells us that the maximum value of the pixels in the "signal" is 198. We also see that the pixel values rapidly change. That is because the signal is extracted from an area where the monkey has a lot of hair. The rapid color change caused by the monkey's hair causes the signal to go up and down quickly. From the previous section, we know that, if a signal's value changes rapidly, then it has high frequency sine signals (components) in it. Now let's see what happens when we press the "LOW-PASS FILTER SIGNAL" button:

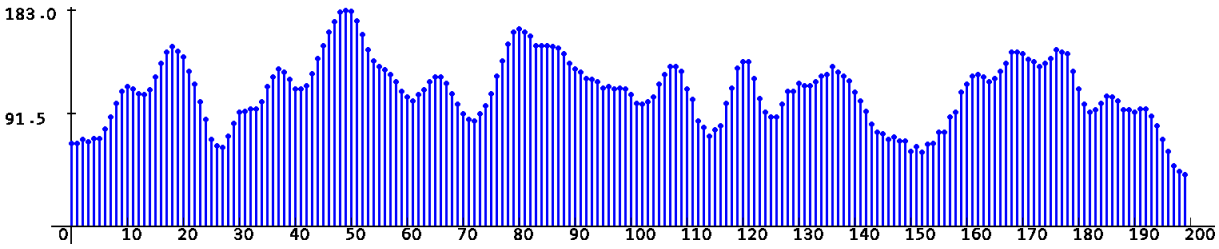


Figure 2.13. The "signal" shown in Figure 2.12 after low-pass filtering.

As we can see from this result, after the low-pass filtering is applied, the signal's values do not fluctuate as rapidly as before. The higher frequency components are suppressed. Now let's take a look at the function that accomplished this:

```
void LowPassSignal()
{
      ICBYTES filter_output, graph;
      ICBYTES filter = { 0.1,0.1,0.2,0.2,0.2,0.1,0.1 };
      Filter_H(signal, filter_output, filter);
      BarChart(graph, filter_output, 300, 2, 0xffffff, 0xff);
      DisplayImage(FRM3, graph);
}
```

In this function, we see that a vector called "filter" is defined with 7 coefficients. Then, the "signal" vector is horizontally filtered by the "Filter_H(.)" function using those coefficients.

Now let's see (Figure 2.14) what happens to the mandrill image when we press the "LOW-PASS FILTER IMAGE" button:

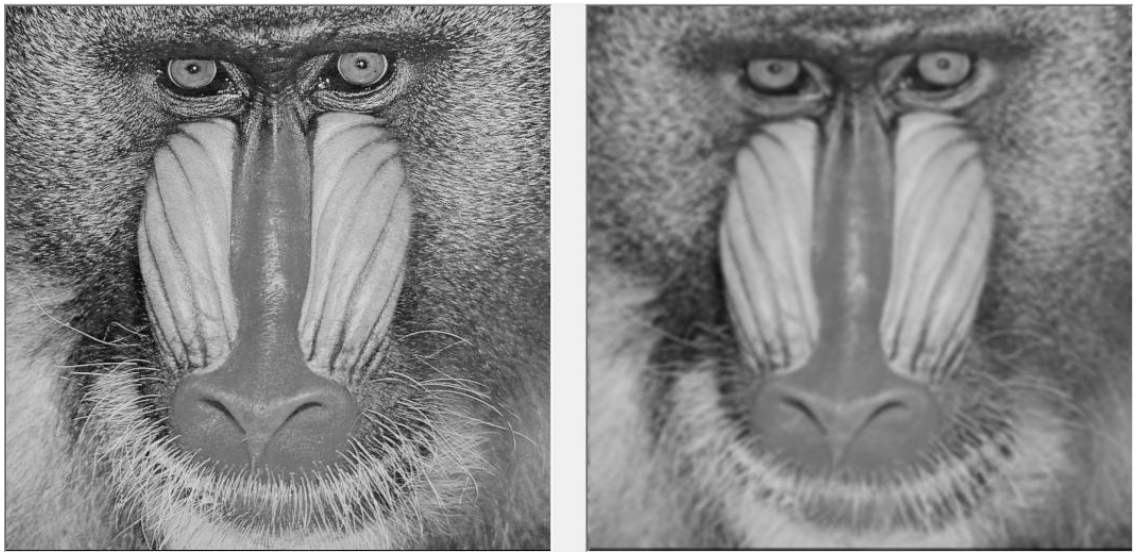


Figure 2.14. Grayscale "Mandrill" image on the left and low-pass filtered version on the right. (Original image has black pixels at the bottom.)

If you look carefully at the mustache of the monkey, you will notice that Figure 2.14 clearly demonstrates how the low-pass filtering makes an image blurry. The effect of suppressing high frequency components in an image is blurriness. You might wonder why anyone would want to make a good image blurry. We wouldn't. However, if you look at the hair of the monkey where we extracted our "signal," you will see that the rapid change of colors between black and white has been reduced as well. Imagine that those rapidly altering colors in the monkey's hair are not from the actual object (in this case the monkey) but are caused by noise. Then, we can say that:

LOW-PASS FILTERING REDUCES NOISE.

The proof of this is that larger objects with less color fluctuations, such as the nose and the eyes are less effected by the filtering. On the other hand, the hair and the mustache got intensely blurry.

So, what is the cause of noise in an image? The main cause is the camera sensor. In a camera, each pixel value is generated by a tiny light-to-voltage converter sensor. In an HD camera there are 1920×1080 (×3 for RGB) tiny sensors that convert to electricity. An approximate function of this conversion is given by this equation: V_{out}=*Light × sensitivity + bias*.

Unfortunately, the fabrication process of a camera sensor is not perfect. As a result, each pixel has different sensitivity and bias values. This means that, even when you take a picture of a perfectly uniform color object under perfectly uniform sun light, each pixel value coming from the camera will be different from each other. That is why industrial machine vision cameras have calibration procedures that can correct this problem through software.

On top of these imperfections, thermal noise, electromagnetic signals coming from objects such as your mobile phone, and discretization effects can add on to that noise. The discretization is a well-known problem for programmers as well. Converting an analog signal to digital is like storing a float value in int. If you type:

```
int x = 155.23897712;
```

then x will contain 155. If your code were like this:

```
int x = 155.93897712;
```

x would still contain 155. The correct way of handling float to integer conversion is:

```
int x;
float f=155.500001;
x = (f + 0.5);
```

In this example, f is correctly rounded to 156 if it is equal or higher than 155.5. Otherwise, x will contain 155. Either way, if this float number were a voltage value coming from camera sensor, it would have been altered from its original value while digitization. These all are reasons why digital images, especially when recorded under dark conditions can contain a lot of noise and this noise may cause miscalculations in your algorithms. The code that low-pass filters the image is shown below:

```
void LowPassImage()
{
      ICBYTES RGB, grey, outp;
      ICBYTES filter = { 0.1,0.1,0.2,0.2,0.2,0.1,0.1 };
      ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
      DisplayImage(FRM1, grey);
      Filter_H(grey, outp, filter, ICB_UCHAR);
      Filter_V(outp, grey, filter, ICB_UCHAR, true);
      DisplayImage(FRM2, grey);
}
```

As you can see from the code, the image is first horizontally, then vertically filtered using the same filter coefficients:

```
Filter_H(grey, outp, filter, ICB_UCHAR);
```

means:

Horizontally filter the "grey" image with the coefficients in "filter" and put the result in the "output" matrix. I want the results in the "output" matrix to be of type **unsigned char**. On the other hand,

```
Filter_V(outp, grey, filter, ICB_UCHAR, true);
```

means:

Vertically filter the image inside the "outp" matrix with the coefficients in "filter" and put the result in the "grey" **image** matrix (That is what the "**true**" at the end is for. True for image matrix and false or no entry for ordinary matrix.) I want the results in the "grey" matrix to be of type **unsigned char**.

Filtering the image in both directions using the same coefficients means actually two-dimensional (2-D) filtering with this matrix:

$$M = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.2 \\ 0.2 \\ 0.2 \\ 0.1 \\ 0.1 \end{bmatrix} \times \begin{bmatrix} 0.1 & 0.1 & 0.2 & 0.2 & 0.2 & 0.1 & 0.1 \end{bmatrix}$$

$$M = \begin{bmatrix} 0.01 & 0.01 & 0.02 & 0.02 & 0.02 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.02 & 0.02 & 0.02 & 0.01 & 0.01 \\ 0.02 & 0.02 & 0.04 & 0.04 & 0.04 & 0.02 & 0.02 \\ 0.02 & 0.02 & 0.04 & 0.04 & 0.04 & 0.02 & 0.02 \\ 0.02 & 0.02 & 0.04 & 0.04 & 0.04 & 0.02 & 0.02 \\ 0.01 & 0.01 & 0.02 & 0.02 & 0.04 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.02 & 0.02 & 0.02 & 0.01 & 0.01 \end{bmatrix}$$

We must avoid using 2-D filtering as much as we can because when we do horizontal or vertical (1-D) filtering we perform 7 multiplications and additions per pixel. We do it twice so it is 14 multiply-add instructions per pixel. Bu if we perform 2-D filtering using the matrix M shown above, then we have to perform 49 multiply-add instructions per pixel. Therefore we, must try to decompose our 2-D filter into two matrices, one horizontal and one vertical if we can.

2.6. Wrong Coefficients

In the previous example, the sum of the coefficients of the filters are always one. You must make sure that the coefficients of a matrix are between 0.0 and 1.0 because all of the C++ variables have a limit and some of them have very small limits. Most of the time, the pixels or digital signals such as sound and music will be stored in either unsigned char (byte) or 16-bit short integer formats. The maximum value that an unsigned char can hold is 255. Therefore, if your filter generates values larger than 255 and you try to store them in an 8-bit unsigned char, C++ will store only the 8 least significant bits of that value. For example, if you write this line:

```
unsigned char B = 300;
```

you will be actually storing 44 inside the variable "B" which is a far darker pixel value then 255. This is called "overflow" in programming terminology. Now let's alter the coefficients of our filter slightly, and perform the filtering again:

```
void WrongFilter()
{
      ICBYTES RGB, grey, outp;
      ICBYTES filter = { 0.1,0.1,0.3,0.3,0.3,0.1,0.1 };
      ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
      DisplayImage(FRM1, grey);
      Filter_H(grey, outp, filter, ICB_UCHAR);
      Filter_V(outp, grey, filter, ICB_UCHAR, true);
      DisplayImage(FRM2, grey);
}
```

As you can see, we changed all of the 0.2 coefficients to 0.3. The sum of the coefficients is now 1.3. The output of this filter is shown in Figure 2.15.

Figure 2.15. Grayscale "Mandrill" image after being low-pass filtered with wrong coefficients. The dark areas are due to the overflow of unsigned char.

How can we correct this problem? The obvious choice would be to reduce the sum of filter coefficients to one but sometimes you may need a special type of filter to have a sum of coefficients that are more than one or less than zero. For those cases you can use the function "RoundFloat2Byte(.)" as shown below.

```
void Rounding()
{
      ICBYTES RGB, grey, outp;
      ICBYTES filter = { 0.1,0.1,0.3,0.3,0.3,0.1,0.1 };
      ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
      DisplayImage(FRM1, grey);
      Filter_H(grey, outp, filter, ICB_FLOAT);
      RoundFloat2Byte(outp, grey);
      Filter_V(grey, outp, filter, ICB_FLOAT);
      RoundFloat2Byte(outp, grey);
      DisplayImage(FRM2, grey);
}
```

In this function, we still use the wrong filter coefficients but in order to avoid the overflowing, we ask the filter functions to return the output in a **float** matrix:
`Filter_H(grey, outp, filter, ICB_FLOAT);`
Float can hold much larger values than 255 therefore, there is no danger of overflow. Then we use the RoundFloat2Byte(.) function to round the **float** values to **unsigned char** (byte). This function clamps the float values so that they are neither less than zero nor larger than 255. On top of that, while converting float values to integer, it does not simply take the integer part like the C++ compiler does, instead it rounds them by adding 0.5. In other words, if you have a 155.93 in the float matrix, the corresponding value in the unsigned char (byte) matrix is 156, not 155. This makes those filter calculations **scientifically more accurate**. The output of this function is shown in Figure 2.16.

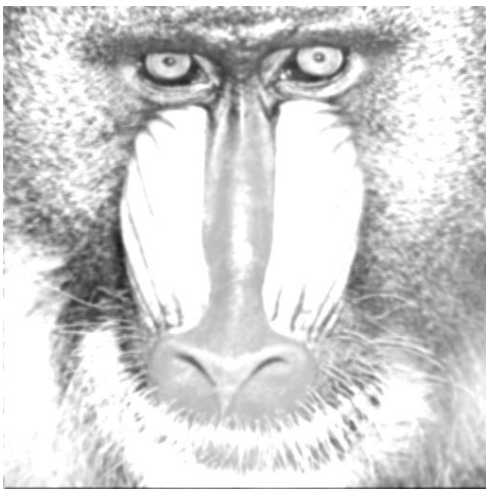


Figure 2.16. The result of filtering with wrong coefficients using byte clamping.

2.7. High-Pass Filtering

In this experiment, we shall demonstrate the effects of high-pass filtering. In the previous section we discovered that low-pass (LP) filtering makes an image blurry. Using this logic, we can infer that the high-pass (HP) filter works the opposite way and makes a blurry image sharper. In this example we will first make the mandrill image blurry and then try to recover the original image. We made the following changes to the "Rounding(.)" function:

```
void Rounding()
{
      ICBYTES RGB, grey, outp,lowpass_out;
      ICBYTES filter_low = {0.1,0.1,0.2,0.2,0.2,0.1,0.1};
      ICBYTES filter_high = {-0.1,-0.1,0.0,0.4,0.0,-0.1,-0.1};
      ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
      Filter_H(grey, outp, filter_low, ICB_UCHAR);
      Filter_V(outp, grey, filter_low, ICB_UCHAR, true);
      lowpass_out = grey;
      DisplayImage(FRM1, grey);
      Filter_H(grey, outp, filter_high, ICB_FLOAT);
      Filter_V(outp, grey, filter_high, ICB_FLOAT);
      grey *= 25.0;
      RoundFloat2Byte(grey, outp);
      outp += lowpass_out;
      DisplayImage(FRM2, outp);
}
```

In this function, we used the same "correct" filter coefficients to make the image blurry, and kept a copy of it in the "lowpass_out" matrix. Then we filtered the image with a high-pass filter whose coefficients are stored in the "filter_high" vector. Note that the sum of the coefficients of this filter is zero. It has both negative and positive coefficients as shown in Figure 2.17-a.

(a) (b) (c)

Figure 2.17. (a) HP filter coefficients. (b) Blurred image. (c) Recovered image.

High-pass filter coefficients contain both negative and positive numbers and usually their sum is zero. This means that the output of an HP filter contains much lower values then the output of an LP filter. That is why we multiply the output of the HP filter (grey*=25.0) with 25 and then add this result to the input image, which actually is the LP filtered image. Note also that the input image to the Filter_V(.) function is of type float. The filter functions can input and output any kind of matrix, however, the filter coefficient vector must always be of type **double**.

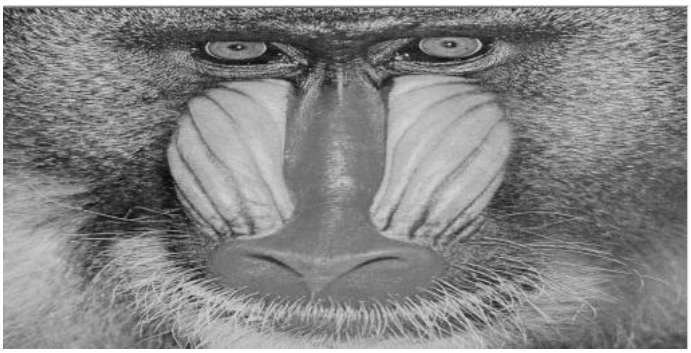
2.8. Down Sampling

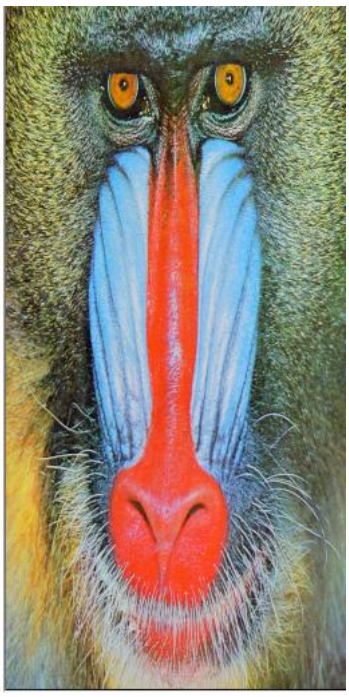
A signal can be 2:1 down sampled by selecting every even or odd sample. That way the amount of data to be processed can be reduced at the expense of information loss. Sometimes, robot vision algorithms may have enough information even in a down sampled image to reach the correct results. Under such circumstances, down sampling speeds up the process by reducing the amount of data to be processed. For images however, it is better to take the average of two neighboring samples.

I-See-Bytes library allows the user to down sample signals and images 2:1 both in x and y directions using DownSampleImg_X(.) and DownSampleImg_Y(.).

The following functions' output is shown below:

```
void DownSample()
{
  ICBYTES RGB, RGBhalf,grey, greyhalf;
  ReadImage("mandrill.bmp", RGB);
  DownSampleImg_X(RGB, RGBhalf);
  Luma(RGB, grey);
  DownSampleImg_Y(grey, greyhalf);
  DisplayImage(FRM1, greyhalf);
  DisplayImage(FRM2, RGBhalf);
}
```





3. Thresholding & Segmentation

3.1. Raw Image Types

In the I-See-Bytes library there are many types of raw image types and more of them will be added in the future versions. However, the most important types are hardware compatible ones. There are two types of hardware: input and output. The input hardware are cameras, scanners etc. that generate those images. The output hardware are the ones that you can display those images. Those are mainly computer screens and printers.

Most of the time, a computer vision engineer spends his/her time in front of a computer displaying the results of algorithms on different type of images that has been saved on the hard disk. Therefore, it is important to know the types of raw image formats that a video card can display. The native image format of a video card is also the type of image that it can display fastest and that is another important reason for us to learn these formats because we wish to design the fastest software.

If you magnify a computer monitor (screen) or a TV as shown in figure 3.1, you will see that the surface of the screen consists of tiny colorful dots. Dots that are red, blue and green:

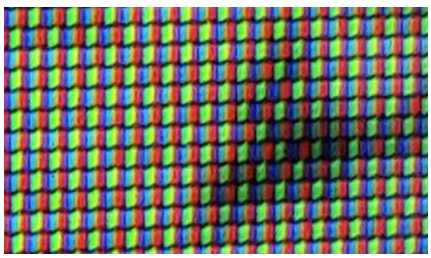


Figure 3.1. Magnified view of a computer screen.

The reason for that is because the different combinations of those three colors can produce any color that we can see (or we think that we see). The human eye has exactly the same type of receptors (neurons) that sends signals to our brain when they see those colors. This has a profound meaning: When we think that we see the color yellow, we might not be seeing a yellow light whose wavelength should be between 575 and 585 nano meters (nm) at all. We might be seeing a combination of red (620 nm) and green (550 nm). Our brain thinks that it sees a yellow object when the light coming from that object has both red and green light in it. Or it may really have yellow light coming out. It doesn't make no difference to us. Consequently, that is how we designed our electronic devices. If images are being recorded or displayed, they are always expressed as combination of these three colors: Red, Green and Blue. RGB for short.

If you have two small lights, red and green, and if you put them next to each other far away enough from yourself, you will see a single yellow light. If you repeat this experiment using combinations of red, green and blue lights, you will see the colors as shown in figure 3.2.

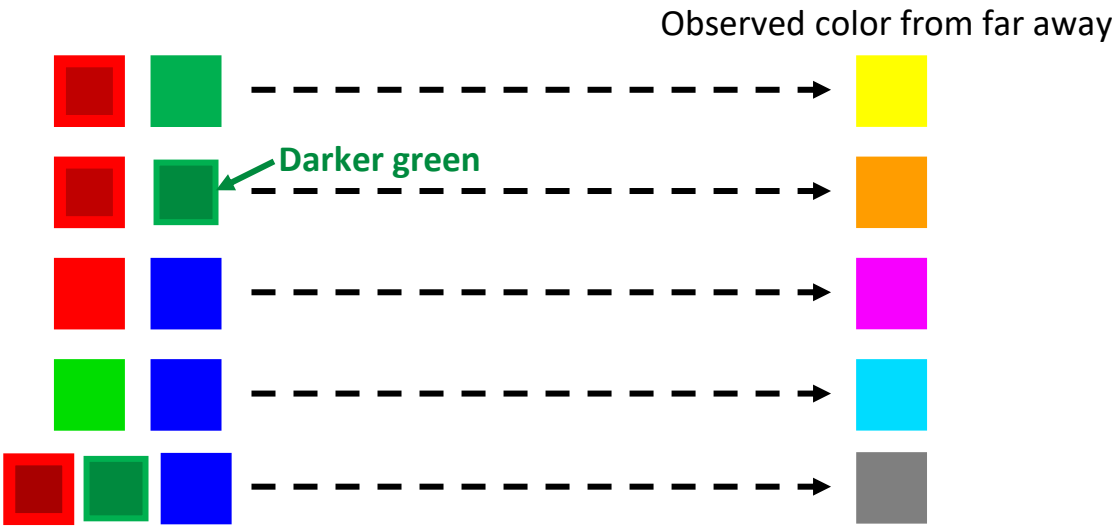


Figure 3.2. Different combinations of lights when observed from far away are seen as different colors.

If all three colors, RGB, have the same amount of brightness, then the observed light color will be shades of gray. For pixels, the amount of brightness is set by an unsigned integer which usually has a value between 0 and 255. An unsigned byte is enough to hold these values, therefore, 3 bytes are enough to create all colors that a human eye can distinguish.

Figure 3.3. Three bytes are required to set (select) the color of a pixel.

Every pixel in a color image must have 3-byte values assigned to it in order to create life like pictures. We mentioned that digital pictures are represented as matrices. So how do we represent different colors? Each color is called a channel and the matrix can have channels as in OpenCV for each color which is a bit confusing and hard to deal with. Or, as in I-See-Bytes, we can represent the color as the third dimension as show in Figure 3.4.

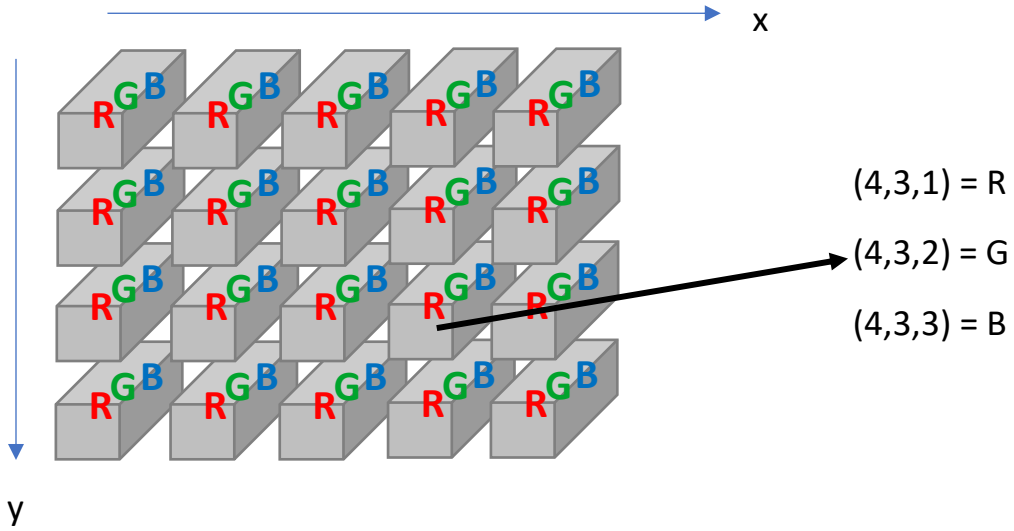


Figure 3.4. In the I-See-Bytes library, color is the third dimension of a digital image.

In order to create such an image (which is also called a 24-bit image), we can use the following commands:

```
ICBYTES RGB;
CreateImage(RGB, 200, 200, 3, ICB_UCHAR);
```

These commands will create a 200×200 image with 3 bytes (or 24 bits) allocated for color representation (depth). If we wanted to set the brightness of the green component of the pixel on the 4th column and 3rd row to 155, we can simply write: RGB.B(4,3,2)=155. (.B for “Byte” or uchar).

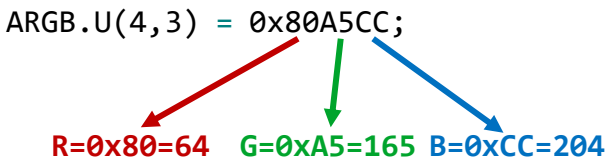
Another popular raw image format is 32-bit image. Addition to the three bytes of red green and blue channels, there is an alpha channel which may be used for different purposes such as transparency or extra brightness. Such formats can be abbreviated as ARGB (A for alpha). To create such an image in I-See-Bytes:

```
ICBYTES ARGB;
CreateImage(ARGB, 200, 200, 4, ICB_UCHAR);
```

you can either define the matrix as 3-dimensional with color dimension having 4 bytes (4×8 bit = 32 bit) or,

```
CreateImage(ARGB, 200, 200, ICB_UINT);
```

Since unsigned integer (UINT) is 32 bits long, this 2-dimensional matrix still provides 32-bit color depth for your pixels. For this case, you need to access the elements of the matrix using "U" (short for UINT) as shown below:



This expression sets the red component of the 4th column and 3rd row pixel to 64, the green component to 165 and the blue one to 204 in one single assignment. In order to achieve that, we use the hexadecimal number system because we can identify the values of each channel by looking at the digits of an hexadecimal number. In C/C++, any expression starting with 0x is treated as hexadecimal. If we don't want to use hexadecimal system we can do the following:

```
ARGB.U(4,3) = 8431052;
```

That expression will set the RGB components to 64, 165 and 204 also. Unfortunately, we cannot see that by simply looking at the number 8431052.

How about the 4th component, alpha? For the moment we shall not use that component. If we are not going to use the alpha component, why do we use 32-bit images? Because we can set the RGB components with a single instruction instead of three and that is %300 faster. Note that the sequence of RGB can be different for different hardware, it can be BGR or ABGR or some other combination. Do not assume that every video card, camera or image format use the same sequence of colors in memory.

3.1.1. Grey Level Images

What if we want to create black and white images that has only shades of grey? One way to do that is obviously to use RGB-24 or ARGB-32 images and set all colors of a pixel to the same level. For example, if you set R=128, G=128 and B=128, then you will obtain the grey shade that is in the middle of the grey spectrum. But that would be wasting memory. If every component of each pixel has the same value then there has to be a way to create a matrix that holds only one value instead of keeping three copies of the same value. That method is:

```
ICBYTES GRAY;
CreateImage(GRAY, 200, 200, ICB_UCHAR);
```

Simply, a 2-dimensional matrix with a depth of 8-bits will give you a gray level image. You can set the gray level of a pixel by writing:

```
GRAY.B(4,3) = 128;
```

“B” is for byte, which is the same as unsigned char in C/C++. To convert a color image to grey level, use the Luma(.) function:

```
ICBYTES RGB,GRAY;
CreateImage(RGB, 200, 200, 3, ICB_UCHAR);
//..
//do something with the RGB then convert it to grey level:
Luma(RGB,GRAY);
```

You can use both 24-bit and 32-bit images as an input to the Luma(.) function. If you send an already gray level image by accident you will receive a copy of it.

3.2. Color Thresholding

Thresholding is the process of labeling pixels belonging to target objects or regions. For example, you might need to label pixels that belong to a product or the background. In a factory where machine parts are manufactured, you should inspect these parts before selling to your customers. If you sell defective machine parts to your customers, they will complain, return the parts back to you, or abandon purchasing your products all together. That is why every manufacturer perform quality control on their products before selling them to customers. But those customers that buy those machine parts are usually companies themselves and if they don’t trust the seller’s quality, they can set up a quality inspection department themselves to ensure the quality of the machine parts they purchase. Otherwise, the products they manufacture using those machine parts they bought would be of low quality.

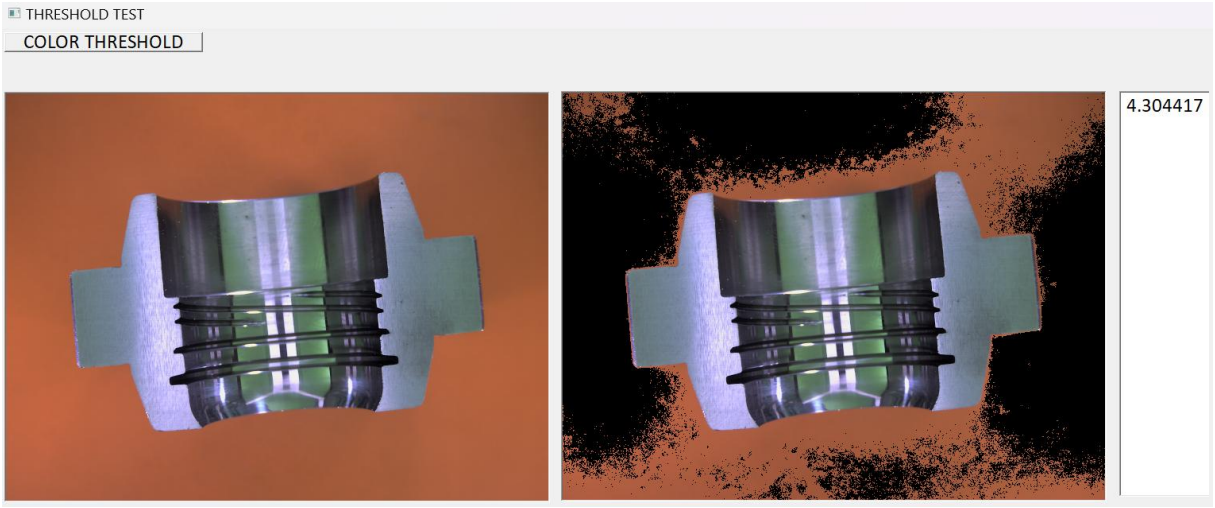


Figure 3.5. A machine part photographed on top of an orange background and the first attempt to threshold the pixels belonging to it.

Figure 3.5 shows a machine part that we wish to inspect. The metal object is located on top of an orange background and photographed. The actual digital image has much higher resolution but for our purpose this resolution is adequate. If we know that the object is located around the middle of the image frame, then we can use this trick to learn the background: We can learn the colors at small sections of the corners because we know that they always belong to the background.

If you look at this picture carefully, you will notice that the metal part has bluish-grey color. Steel is silvery shiny, but it also has tendency to be slightly blue. That is why we hear the “blue steel” phrase very often. Who ever chose this background, wanted a color opposite to blue so that the thresholding process can be performed more easily. We also see green reflections on the machine part meaning that an opaque, green light source has been used to illuminate the object. This means that we can use the color difference between the background and the object to label the pixels. In order to do that, we write a program that calculates the average yellow (red+green) and blue components of the 50×50 pixel portion of the top left corner of this image:

```
ICBYTES RGB,masked;
ReadImage("machine_part.bmp", RGB);
CreateImage(masked, RGB.X(), RGB.Y(), RGB.Z(), ICB_UINT);
float blue=0, yellow=0, dark_thresh, color_thresh;
int x, y;
unsigned char* pix;
for (y = 1; y <= 50; y++)
{
      pix = (unsigned char*)RGB.U_[y];
```

```
	for (x = 1; x <= 50; x++)
	{
		//		green		red
		yellow += pix[x * 4 + 1] + pix[x * 4 + 2];
		blue += pix[x * 4];
	}
}
```

In this code, we first load the "machine_part.bmp" as a 32-bit color image matrix called "RGB." Then we create another 32-bit blank (empty) image exactly the same size as that one. Using the "for" statements, we sum the green and red components of the pixel values and store the sum inside the variable called "yellow." In order to do that, we type cast the unsigned int values of 32 bits to unsigned char using the following code line:

```
pix = (unsigned char*)RGB.U_[y];//the first pixel of every row
```

We also sum the blue components and store them in another variable called "blue." That makes a total of 50×50=2500 pixels. You would think that such an amount would be enough to give us the average strength of yellow (or orange) and blue components of the background. Therefore, we set our threshold for pixel color as follows:

```
color_thresh = yellow / blue;
ICG_printf(MLE,"%f\n", color_thresh);
```

This gives us 4.304417. Now that we know the ratio of the orange to blue components of background pixels, we can use this knowledge to label the entire pixels of the image as background and foreground (metal object). We know that the orange component of the metal is much lower than the background but the blue component is much higher. Therefore, if we scan the entire image and calculate orange/blue ratio for each pixel, we can decide if that pixel belongs to the orange background or the metal foreground. The following code does exactly that:

```
for (y = 1; y <= RGB.Y(); y++)
{
	pix = (unsigned char*)RGB.U_[y];
	for (x = 1; x <= RGB.X(); x++)
	{
		//		green		red
		yellow = pix[x * 4 + 1] + pix[x * 4 + 2];
		blue = pix[x * 4];
		if ((yellow / blue) > color_thresh) masked.U_[y][x] = 0;
		else masked.U_[y][x] = RGB.U_[y][x];
	}
}
```

```
DisplayImage(FRM1, RGB);
DisplayImage(FRM2, masked);
```

This code calculates the yellow/blue ratio for each pixel and if the ratio is larger than the threshold calculated in the previous step than that must be a background pixel and the corresponding pixel (the pixel with the same coordinates) in the "masked" image is set to color black. On the other hand, if the threshold is not higher, then it is considered a foreground pixel and the corresponding pixel's color in the masked image is set to the color of RGB's image. This way we hope that all the orange background turns black (0) in the "masked" image and only the metal foreground remains.

Unfortunately, looking at the resultant "masked" image in Figure 3.4, we see that nearly half of the orange foreground pixels are still there. Where did we go wrong? If we look at the original image carefully, we will notice that the background color is not exactly uniform. There are slightly darker and lighter areas caused by nonuniform illumination. As a result, our threshold is too high for some regions. In that case, we can try to lower the threshold just a little to see what happens. The original threshold value is 4.304417. If we subtract 0.8 from this threshold value:

```
color_thresh -= 0.8;
```

And run the last part of the code again. We will get the image in Figure 3.6. some people might think that using a threshold is wrong and the threshold value should be adaptive or pattern classifier should be used. Those are ignorant people who do not know or understand the math behind these algorithms. First of all, when we calculated the average yellow and blue components of the 50×50 pixel portion of the top left corner, that was adaptively learning the threshold.

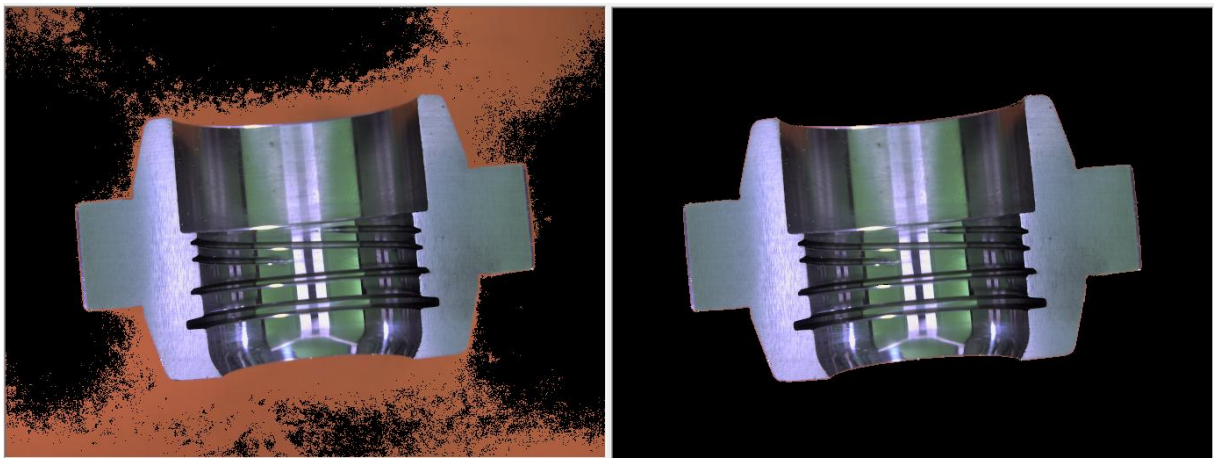


Figure 3.6. The first attempt to threshold the pixels belonging to the metal object (left), and the second one with slightly lowered threshold (on the right).

Secondly, a pattern classifier is IN FACT a threshold machine. A pattern classifier receives many features as input and it determines the optimum thresholds based on those features. If there were only one feature available, then any pattern classifier would be reduced to a single threshold. But people who do not understand the math behind the pattern classifiers are unaware of that.

The important thing is to be efficient. And the best way to be efficient is to find the one perfect feature that solves your problem. That is the philosophy behind good engineering. Simple, fast and inexpensive. Unfortunately, perfect features do not come by easy and the reader should not think that the feature we used here, yellow/blue ratio, is a perfect feature. There are many ways to improve up on it. Our goal here is to demonstrate the basic concepts of color thresholding.

3.3. Grey Level Thresholding

Figure 3.7 shows a poor-quality photograph of a book page. If we wanted to recover the letters from the background, what type of pixel threshold should we use? The problem with grey level images is that there are no colors such as yellow and blue whose ratios we can use to create a threshold. All colors are grey. In that case, we have no other option but to use the pixel values as threshold. In a grey level image pixels values may vary between 0 (black) and 255 (white). If we create a histogram of pixel values perhaps the histogram might give us an idea.

Figure 3.7 also shows the histogram of this image. A histogram is a count of occurrence of a pixel value in that image. For example, according to the histogram, neither full black (pixel value 0) nor full white (pixel value 255) occurred in this image. On the other hand there are 967 pixels with a brightness of 146.

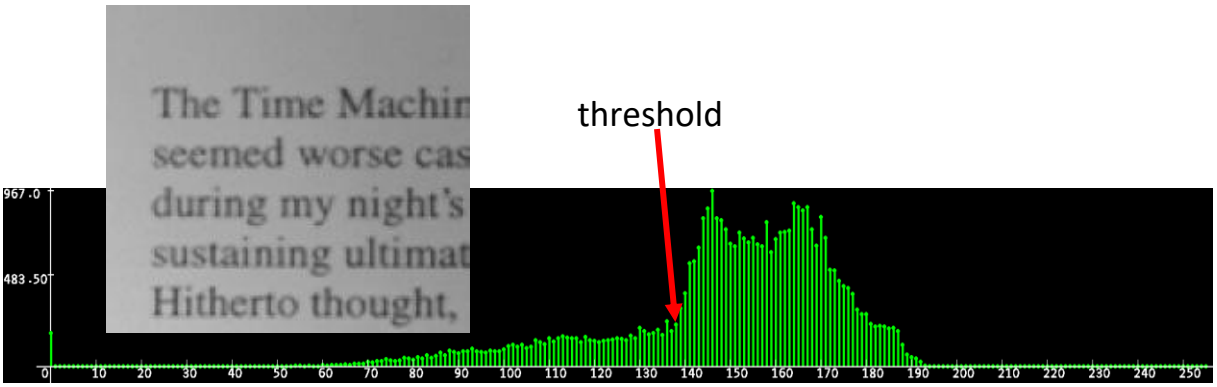


Figure 3.7. A poor-quality photograph of a book page and its histogram.

The code that generates and displays the histogram is shown below:

```
ICBYTES RGB, grey, histo, histograph;
ReadImage("textsample.jpg", RGB);
Luma(RGB, grey);
GrayHisto(grey, histo);
BarChart(histograph, histo, 300, 3);
DisplayImage(FRM1, RGB);
DisplayImage(FRM3, histograph);
```

This code loads the photograph of the book page and then converts it to a grey level image matrix using the function Luma(.). Then we call the GrayHisto(.) function to create a vector that contains the grey level histogram of the image. We are already familiar with the BarChart(.) function (Chapter 2).

Ideally, when we look at a histogram, we would want to see two hills as far away from each other as possible. These two hills are created by the foreground and background pixels. If that were the case, we could select a point equally distant from both peaks and call it the threshold value. Unfortunately, in our histogram, there is only one hill with a rugged peak. The location of the hill is close to the right (bright) side therefore, we can assume that the hill belongs to the background pixels which are relatively bright. In that case we can select the left foot of the hill as the threshold point which is around 138. That point is shown with a red arrow in Figure 3.7. In order to see what happens when we threshold this image using 138 as the threshold, we use the following program:

```
ICBYTES RGB, grey, masked;
ReadImage("textsample.jpg", RGB);
Luma(RGB, grey);
GrayThresh(grey, masked, 138);
DisplayImage(FRM2, masked);
```

The above code uses the function GrayThresh(.) to threshold the gray level image. In the previous example for color thresholding, we had to write a nested for-loop to scan all of the pixels one by one. The GrayThresh(.) function does exactly that for us but works only on grayscale images. The output of this code is shown in Figure 3.8-b.

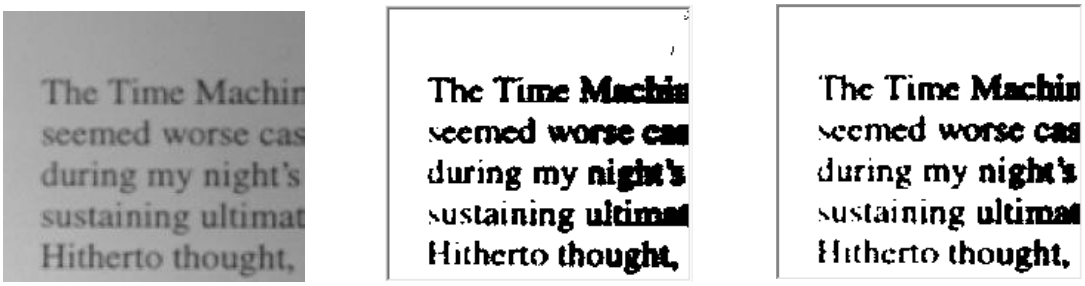
You may have noticed the difficulty of choosing the correct threshold value. In this image, despite the fact that the foreground (letters) and the background (paper) pixels are clearly distinguishable by the human eye, we still had some difficulty choosing a threshold value. What if the image were more complicated and it looked like the foreground and the background colors were so close to each other that they looked like they merged together? Is there a way to automatically select the threshold value? Fortunately, many scientists worked

on this problem and they developed different methods. One of the most popular methods is Noboyuki Otsu's method. This method is implemented in I-See-Bytes library and should be used as follows:

```
ICBYTES RGB, grey, masked;
ReadImage("textsample.jpg", RGB);
Luma(RGB, grey);
int thresh = Otsu(grey);
ICG_printf(MLE, "%d\n", thresh);
GrayThresh(grey, masked, thresh);
DisplayImage(FRM1, grey);
DisplayImage(FRM2, masked);
```

In the above code, the grey level image matrix (8-bit color depth) is sent as an argument to the Otsu(.) function and the function returns an integer threshold value. Then we use this value as an input argument to the GrayThresh(.) function along with the greyscale image to obtain the binary image shown in Figure 3.8-c. The threshold value returned by Otsu(.) is 132 which is very close to the one we selected (138). Despite the fact that our threshold was only 6 levels higher than the one Otsu(.) returned, the improved result can be clearly seen between Figure 3.8-b and 3.8-c.

After thresholding, the images have only two pixel levels: 0 and 255. Those types of images are called binary images. When we look at those binary images, we see that the letters on the lefts side are missing pixels and the letters on the right side have extra pixels that make them as if they are bold style. The reason for that is because the illumination of the page is nonuniform. While the left side of the image has more illumination, the right side has less. Because we apply the same threshold to the entire image, we have such issues. The smart thing to do would have been to divide the image to sections and apply Otsu's threshold algorithm to those sections as if they are separate images. The result would have been much better.



(a) (b) (c)

Figure 3.8. (a) A poor-quality photograph of a book page. (b) Result of thresholding with our choice (138). (c) Result of Otsu's method (132).

3.4. Pixel Segmentation

Consider the images shown in Figure 3.9. The first one shows the blood cells under a microscope. The second one shows cherries on a table and the third one is enlarged (zoomed) view of rice grains. Imagine that you need to write a program that counts these objects. One simple method may be to threshold the images, count the foreground pixels and then divide the number of foreground pixels to the average size of the objects. Although that would give us a very good estimate, unfortunately, both the "Blood cells" and the "Rice Grains" have very different sizes in terms of pixels and that approach would lead to wrong answers.

(a) (b) (c)

Figure 3.9. Three different images that we wish to count the objects in. (a) Blood Cells. (b) Cherries. (c) Rice Grains.

A better way to solve this problem is to group the pixels belonging to objects. If objects are not touching each other, then all the pixels that touch each other must belong to the same object. In that case, if we can count how many groups of connected pixels that are, then we will know exactly how many objects there are in the image. Figure 3.10 shows magnified pixels of a binary image with two separate objects on it. If we could group (label) these pixels based on which object they belong to, then we would know exactly how many groups of pixels, or in other words how many distinct objects there are in the image.

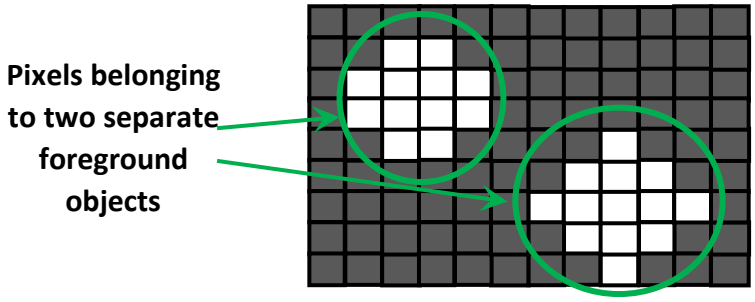
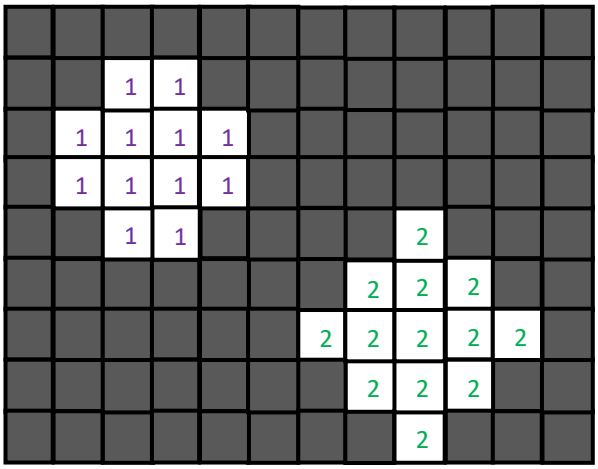


Figure 3.10. Magnified binary image with dark pixels as background and white pixels as foreground.

Segmentation process does exactly that. It either labels each pixel based on which object they belong to (Figure 3.11-a) or, it creates a list of pixel coordinates (x,y) for each object (Figure 3.11-b).



List of Pixel coordinates for each object:

1- (3,2), (4,2), (2,3), (3,3), (4,3,), (5,3), (2,4), (3,4), (4,4), (5,4), (3,5), (4,5).

2- (9,5), (8,6), (9,6), (10,6), (7,7), (8,7), (9,7), (10,7), (11,7), (8,8), (9,8), (10,8), (9,9).

(a) (b)

Figure 3.11. Results of binary image segmentation. (a) Pixel labeling. (b) Pixel listing.

Both of these formats are useful for different purposes. For example, pixel labeling allows you to trace the peripheral pixels more easily. On the other hand, listing coordinates allows you to find the center of gravity of the object faster. Therefore, the I-See-Bytes library prepares both of these lists. The function that segments binary images is called SegmentPixels(.). After the segmentation process, this function returns two ICBYTES objects. The first one contains the labels as an unsigned short matrix that is the same size as the image. The second one contains a vector list which contains the coordinates of the pixels of each object. There is also a function that paints these segments using different random colors and that one is called PaintSegments(.). It accepts either the labels or the pixel lists the SegmentPixels(.) prepares.

WARNING: SegmentPixels(.) function always assumes that foreground objects have white pixels and background pixels are dark. When you simply threshold an image, you need to know if your foreground objects are darker or brighter than the background. For example, in Figure 3.9, “Blood Cells” and “Cherries” have bright background. On the other hand, the “Rice Grains” picture has a dark. You need to use the hidden fourth parameter (true/false) of GrayThresh(.) function if you want its result to be color inverted. Default is false so it doesn’t invert. Make it true and it will.

Figure 3.12 shows the results of thresholding and segmentation for the images shown in Figure 3.9. The code that generates cherries is given below:

```
ICBYTES RGB, grey, masked, seg, map, coloredsegments;
ReadImage("cherries.jpg", RGB);
Luma(RGB, grey);
int thresh = Otsu(grey);
GrayThresh(grey, masked, thresh, true);
SegmentPixels1(masked, seg, map);
PaintSegments(map, coloredsegments);
DisplayImage(FRM1, RGB);
DisplayImage(FRM2, masked);
DisplayImage(FRM3, coloredsegments);
```

Extra "true" here inverts the binary image colors

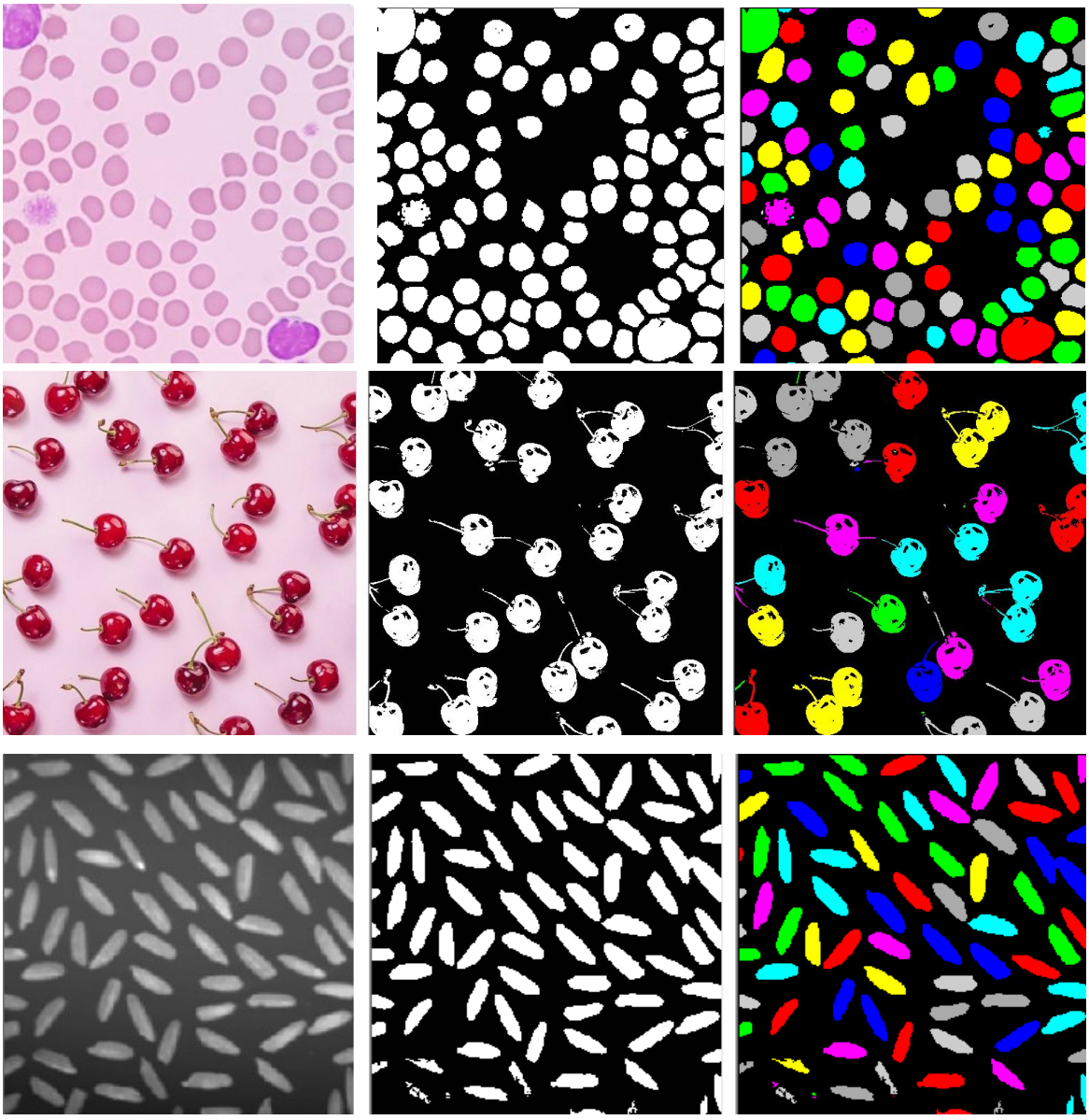


Figure 3.12. Results of grey level thresholding and segmentation. Images of Figure 3.9 (left). Binary images after thresholding with Otsu's method (middle). Segmented and colored versions of binary images (right).

4. Actuator Control

4.1. Definition of an Actuator

In Robotics, any device that causes the robot to move is called an actuator. There are many types of actuators such as electric motors, hydraulic or pneumatic pistons, electrical solenoids and even polymer muscles. The most popular, inexpensive and easy to use actuators are based on direct current (DC) electric motors. These motors can be controlled simply by being turned on and off by a switch. The second most popular type of electric motor is the servo motor which requires a circuitry to control, but is also much more accurate than the DC motor. The third one is the step motor which also requires special circuitry to drive it. Arguably, they can be controlled much more precisely and that is why 3-D printers use step motors.

Other types of actuators, such as hydraulic and pneumatic pistons can also be controlled by simple switches, though there are much more precise ways of controlling them through special driving circuits. Nevertheless, a simple switch that can be turned on and off by a computer is the most versatile actuator control device that can control DC motors, electric solenoids, hydraulic and pneumatic valves.

4.2. Relay Control

The light switches that you see on the walls are operated mechanically meaning that you have to apply a mechanical force, or energy to turn it on and off. What if you wanted a switch that can be controlled using electricity? A relay is a switch that can be turned on and off by supplying it with low amounts of electrical power such as 6 or 12 volts. Using such low voltages, a relay allows you to control voltages such as 120, 220, or even 1000 volts and dozens of Amperes. In other words, by supplying it with a power in the range of 10-100 milli watt range, you can control thousands of watts. You can not only control pretty much any size electric motor but also appliances such as ovens, refrigerators and washing machines.

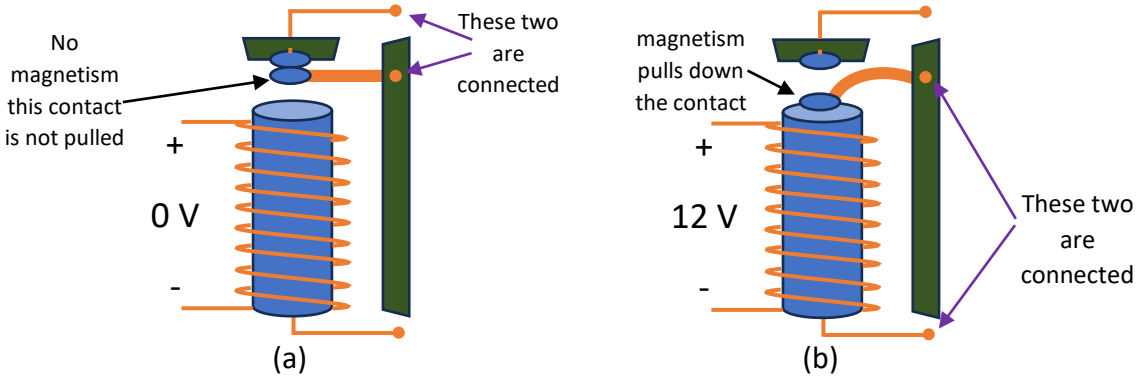


Figure 4.1. The schematic view of the relay. (a) When there is no power applied, the top two outputs of the relay are connected. (b) When the power is applied the bottom two are connected.

So how does a relay work? The most common type, the electromagnetic relay, works by magnetizing a piece of iron and then making it pull a metal and create a contact as shown in Figure 4.1. Imagine that all the orange lines are copper cables; and contact points and all the blue objects are made of a magnetic metal such as iron. When there is no current flowing through the spring like wire wrapped around the cylindyrical iron core as shown in (a), it doesn't create a magnetic field and everything stays put. The top two contact points are touching each other therefore, they can be considered a switch that is "on."

On the other hand, when we apply 12 volts to the spring like coil as shown in (b), the current through the coil magnetizes the iron core and it pulls the little iron ellips which is connected to the middle output. The thick copper wire that the little elliptical iron core is connected to is stiff and it behaves like a spring and would pull the elliptical iron core upwards if there were no magnetizm. As long as we apply 12 volts, the little elliptical iron core would be attracted to the big cylindyrical and stick to it; creating a connection between the middle and the bottom outputs of the relay. You may find it hard to believe, but before the tranzistor was invented, some of the earliest computers used these relays to create logic circuits.

Another actuator, the electric solenoid, works in a very similar way to the relay, in order to create a simple linear motion/force.

4.3. A Simple Mobile robot

The I-See-Bytes library provides a direct interface to the USB DCTTech Relay card. This is an Open-Source equipment meaning that its design can be obtained on the internet. There are numerous companies in the world that produce these cards and they can be purchased from any electronics hobby shop on the web. An 8-relay board is shown in Figure 4.2 where it is used for controlling the motors of a simple 4-wheel drive robot.

The purple cylinders are lithium-ion batteries. Each of these batteries are 3.7 V and two of them in series (7.4 V) is used to drive the motors. Three of them are used for operating the relays. The relay board itself draws its power from the USB connection which is 5 volts. Unfortunately, relays require at least 9 V to operate therefore we need at least three lithium-ion batteries (3.7 V x 3=11.7 V) to operate the relays. The smaller circuits are the battery charger circuits.

WARNING: Lithium-ion batteries are extremely dangerous to work with. They need special circuits to charge them. If you are not an expert don't use them! **THEY CAN EASILY EXPLODE AND CAUSE FIRE HAZARD!**

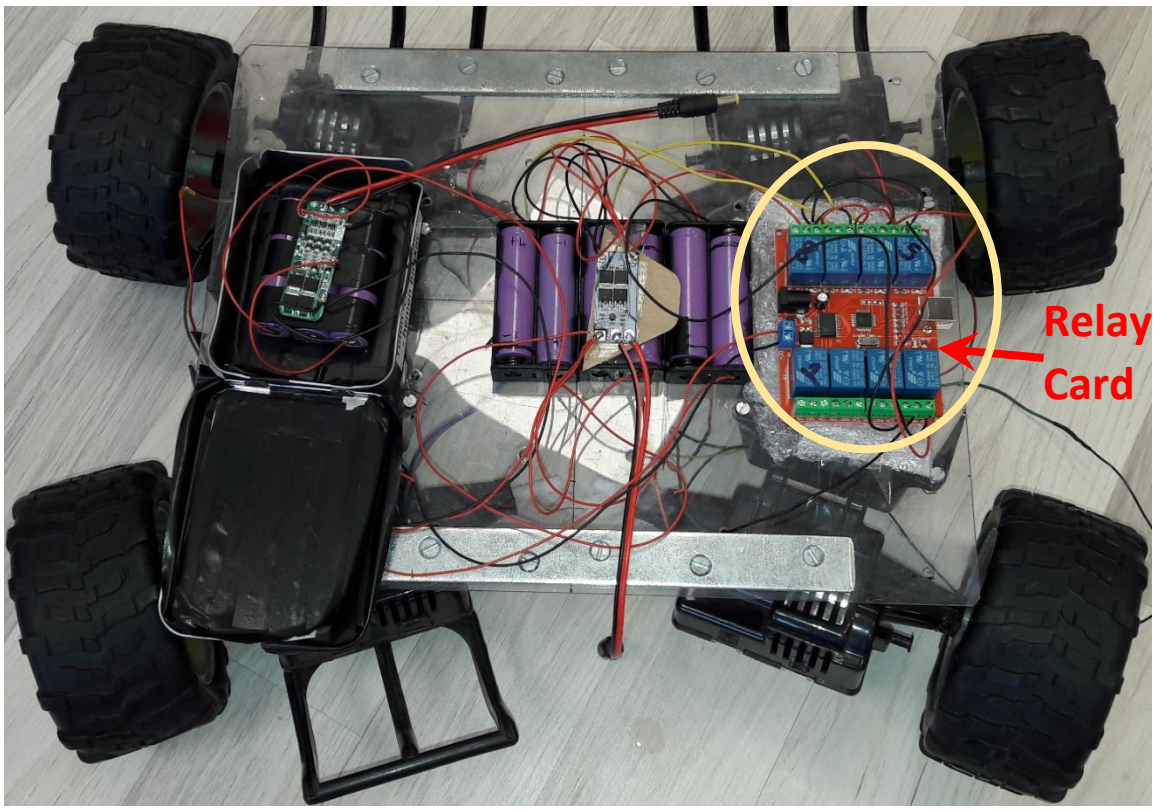


Figure 4.2. Inside of a 4-wheel drive mobile robot. The encircled electronic card is a DCTTech 8 channel USB relay card. Tiny blue boxes on the card are the relays.

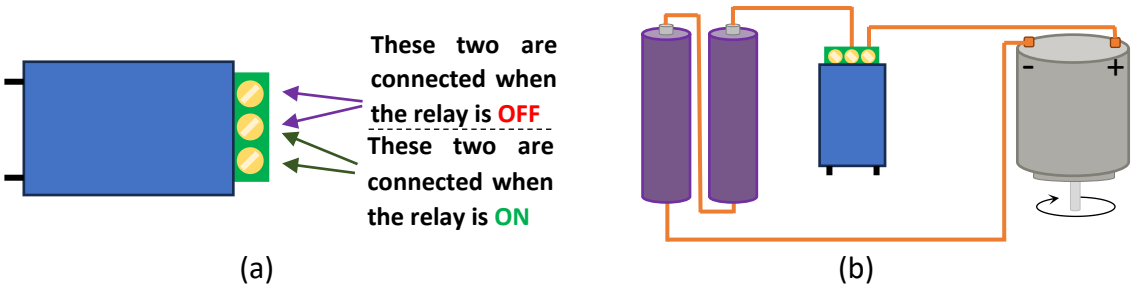


Figure 4.3. (a) The output connections of the relays on the DCTTech relay board. (b) A schematic for controlling the electric motor through the relay.

Figure 4.3-a shows the circuit connections of the relays that are on the DCTTech USB board. When the relay is off, the top two terminals are connected. When the relay is powered on, the bottom two becomes connected. Figure 4.3-b shows how a simple on/off motor control circuit using one of those relays. Unfortunately, if you want your robot to go backwards, the circuit becomes much more complex. When the positive side of the batteries are connected to the positive side of the motor it will push the robot forward. In order to make it revolve backwards you need to alter the polarity and connect the positive pole of the battery to the negative input of the motor. And you need to do that using relays so that the laptop/tablet on top of the robot can command it.

Figure 4.4 shows six of the possible eight moves of a 4-wheel mobile robot the other two are backwards right and left turns. Notice that the wheels on the left side of the robot always turn towards the same direction. That is true for the

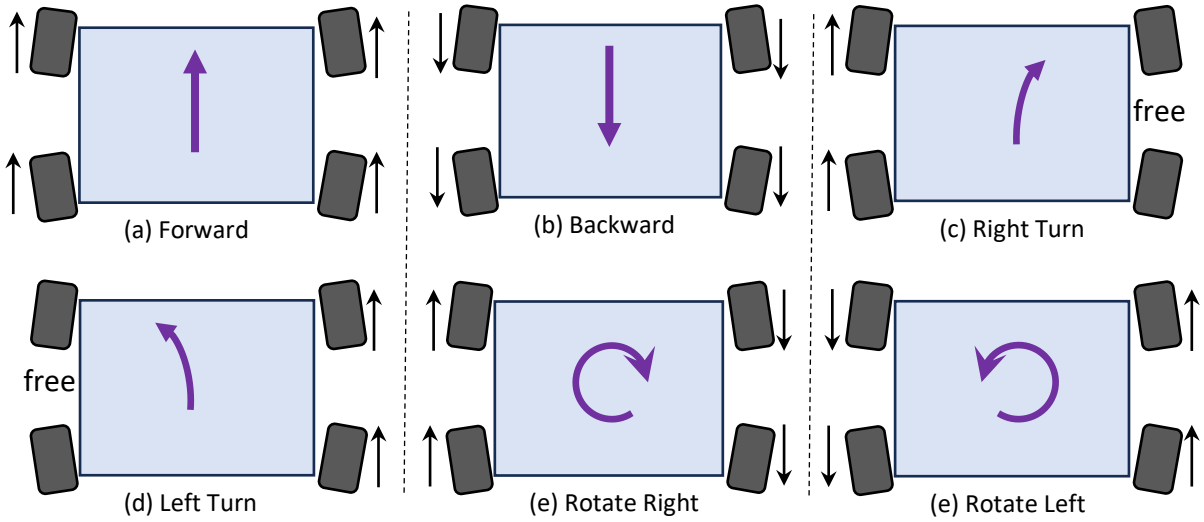


Figure 4.4. Wheel rotation directions for six of the possible eight moves of a 4-wheel drive mobile robot.

right side as well meaning that we can connect these motors in parallel to each

other and we will need only one relay circuit to control the pair on the left and another relay circuit for the right side.

Figure 4.5 shows how the relays should be connected to the pair of motors on each side. The first relay is used for powering up or down. If you turn on relay 3 and keep relay 2 off then the motors will rotate forward (or backward based on how you connected the poles of the motors or which side you are on). If you turn on relay 2 and keep 3 off, then the robot moves backward (opposite way to first case). You have to repeat that for the other side of the robot. Therefore, you will need a total of 5 relays. You can use the rest of the relays for lights or something else.

HOMEWORK: Is there a way to control motors using only two relays?

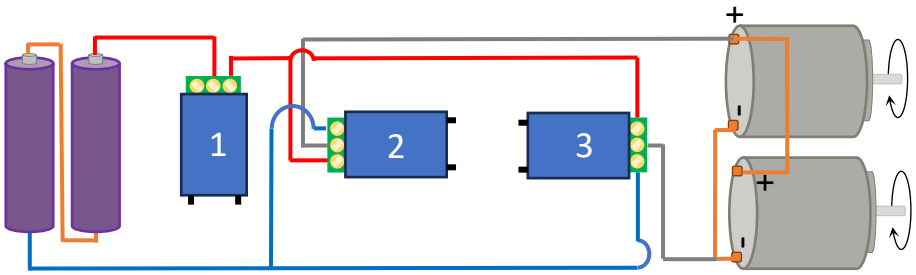


Figure 4.5. (a) The relay connection schematic for each side of the 4-wheel drive robot.

You can use 4 channel relays and a L298 DC motor driver as shown in Figure 4.6 as well.

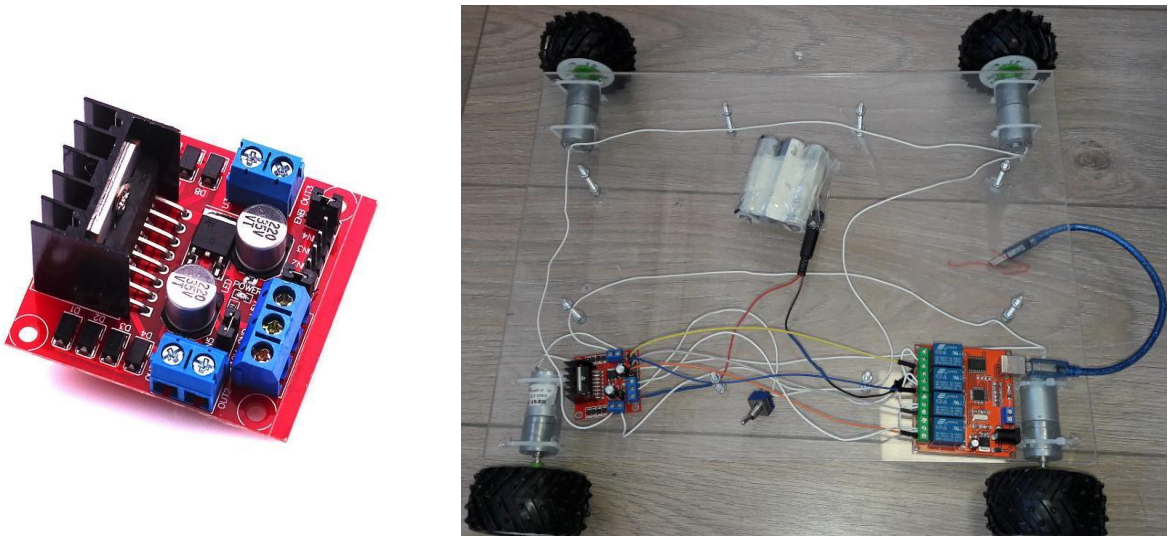


Figure 4.6. (a) The L298 DC motor driver circuit on the left and another mobile robot that uses 4 channel USB relay card and L298 driver.

4.4. A Simple robot Vision Project

With all the things that we have learnt so far, we can build a robot that follows a yellow round object. The figure on the right shows a person holding a yellow disc printed on a paper facing the webcam of a laptop that is resting on a mobile robot. If the laptop is connected to an 8 channel USB relay card which controls the mobile robot, then we could easily write a code that can move the robot towards the yellow round object using the I-See-Bytes library.

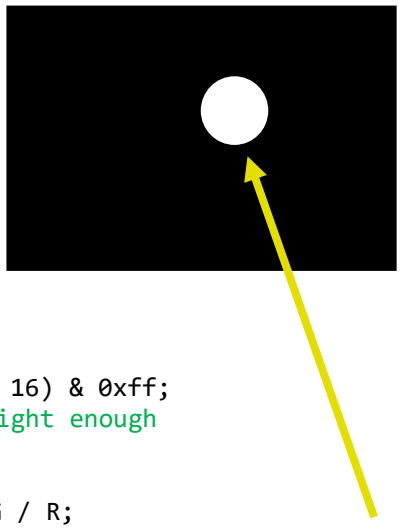
The first thing we need to do is get an image from the webcam, and then threshold the yellow color. We learned how to do that in chapter 3. Then we have to make sure that the object is round so that the robot doesn't follow a person with yellow sweater. Below is the code that performs those tasks:

```
ICDEVICE cam0, relay;
ICBYTES t, RGB1, obj1, seg, map;
void Run()
{
  ICBYTES obj1;
  DWORD dw;
  if (CreateMFCam(cam0, 0) < 0)//Webcam device created here
  {
    ICG_printf("CAMERA 0 UNAVAILABLE!\n");
    return;
  }
  CreateDCTtechRelay(relay, 0);//Creates a device for the relay card
  if (CaptureImage(cam0, RGB1) == 0)ICG_printf("ERROR!!!\n");
  else
  {
    //create an EMPTY image that is half the size of the camera image
    CreateImage(obj1, RGB1.X() / 2, RGB1.Y() / 2, ICB_UCHAR);
    //Below line creates the thread that performs the actual image processing
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)RunThread, NULL, 0, &dw);
  }
}
```

Let's analyze what this code does. This is the function that is called when we press the "RUN" button on the screen. The code that creates the GUI and the buttons will not be shown. As soon as the function begins, it tries to create a MFCam ((Microsoft) Media Foundation Interface) type webcam interface. If it

fails, if shows an error message and terminates. If it succeeds, then it creates a DCTtechRelay type device which is our 8 channel USB relay card. Then it attempts to capture a raw RGB image from the camera to make sure that everything works. If it fails, the function again displays an error message and returns. Otherwise, it creates a thread that continuously performs robot vision tasks. We know that, in Windows operating system, if we want to write programs that continuously work for minutes, we need to write them as a separate thread. Otherwise, the program will freeze. Windows operating system relies on messages and if our program can't process them it will freeze. Now let's take a look at the thread. Keep in mind that the following code is inside a loop:

```
unsigned cl;//holds pixel color RGB
float R, G, B, ratioGB, ratioRB, ratioGR;
if (CaptureImage(cam0, RGB1) != 0)
{
  obj1 = 0;
  DownSampleImg_X(RGB1, t);
  DownSampleImg_Y(t, RGB1s);
  DisplayImage(FRM1, RGB1s);
  for (int y = 1; y <= RGB1s.Y(); y++) {
    for (int x = 1; x <= RGB1s.X(); x++) {
      cl = RGB1s.U_[y][x];
      // Extract (B)lue, (G)reen and (R)ed components
      B = cl & 0xff; G = (cl >> 8) & 0xff; R = (cl >> 16) & 0xff;
      if ((G + R) > 100)//make sure that yellow is bright enough
      {
        B += 1.0; R += 1.0;
        ratioGB = G / B; ratioRB = R / B; ratioGR = G / R;
        if (ratioGB > 2.5 && ratioRB > 2.5 && ratioGR > 0.7) obj1.B_[y][x] = 255;
    }
  }
```



What this code does is, after capturing an image from the camera which in our laptop it is 1280x720 pixels, it down samples the image because we don't need that much resolution to locate a big round yellow object. Remember that we created the EMPTY image "obj1" in the previous function that instantiated this thread function. We first made sure that the obj1 image has a black background (see line that says obj1=0). Then, the code color thresholds the RGB1 image and puts white pixels in obj1 in the same coordinates as the yellow pixels in RGB1. At the end of this threshold, our goal is to obtain a picture in obj1 as shown above (the black rectangle with a round white circle).

However, there can be other yellow objects in the seen and we need to make sure that we target the round object. For example, Figure 4.7 shows a person wearing a yellow T-shirt sitting in front of the camera. Our code eliminated every other color and left only the yellow pixels of the T-shirt. Note

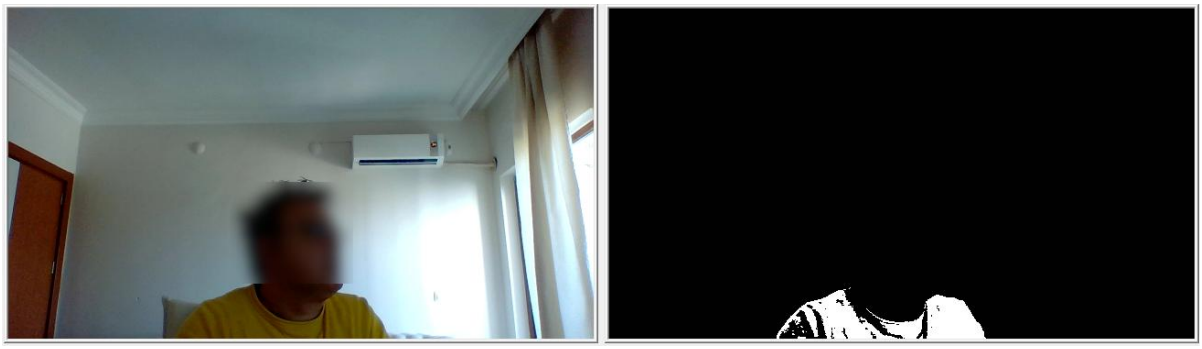


Figure 4.7. A person wearing a yellow T-shirt and the results of color thresholding using the C++ code in this section.

that the algorithm does not get fooled by the similar colored objects such as the brownish-orange door, the beige curtain or the skin color of the handsome person. Since we don't want our robot to follow arbitrary objects but only our round yellow sign, we now need to figure out which yellow objects are round. First, we will use the SegmentPixels(.) function that we learnt in chapter 3:

```
SegmentPixels1(obj1, seg,map);
//seg.Y() gives us the number of yellow objects. If there aren't any there is no
//point in continuing. We wait 100 ms before we get a new image from the camera.
if (seg.Y() == 0) { Sleep(100); continue; }
int area = 0, indx = 0;
//we find the largest yellow object
indx = FindLargestSegment(seg);
//if index is zero it couldn't find any objects we wait before we get a new image.
if (indx == 0) { Sleep(100); continue; }
//Calculate the area of the yellow object in terms of pixels.
area = seg.X(indx) / 2;
//if area is zero it couldn't find any objects we wait before we get a new image.
//Too many security checks right?
if (area == 0) { Sleep(100); continue; }
//we calculate the center (x,y) of the yellow object
for (int x = 1; x < seg.X(indx); x += 2)
{
      xav += seg.u_[indx][x]; yav += seg.u_[indx][x + 1];
}

if (indx > 0 && area>100) //There is a large enough yellow object
{
      xav /= area; yav /= area; //This is the center (xav,yav)
```

Rest of the code shall be an **exercise for the student:**

- How can you figure out if the object is round?
- How do you decide which way to move the robot?
- How do you translate robot moves to relay controls?

Controlling the relays is very easy in I-See-Bytes library. Remember that we created a device using the following commands:

```
ICDEVICE cam0, relay;
CreateDCTtechRelay(relay, 0);//Creates a device for the relay card
```

The zero is for the first card. Ther may be more than one USB cards connected to your computer. In that case you need to enumerate them. To turn on relay no:1 on, you call:

```
CommandRelay(relay, 1, "1");
```

And to turn it back off:

```
Sleep(duration);
CommandRelay(r, 0, "1");
```

The “duration” is an int variable holding amount of the time relay is turned on in terms of milliseconds. 1000 millisecond is 1 second. Don’t forget to close the devices when you are done with them:

```
CloseDevice(cam0);
CloseDevice(relay);
```

4.5. Robot Kinematics

Robot Kinematics is an important terminology and its meaning is the study of robot movement. There are other terms related to this subject such as:

- Degrees of Freedom (DOF)
- Forward Kinematics
- Inverse Kinematics
- Reference Frame

These terms will be explained through the rest of the book in different chapters so as not to overwhelm the student. This is a Robot Vision book and the students are not mechanical engineers. Despite the fact that Electrical Engineering students are more familiar with the matrix math used for teaching these concepts, Computer Science students may not be familiar with them depending on the curriculum of the university they graduated.

Since the majority of the students reading this book is expected to be Computer Science students, subjects related to mechanics, electronics their relationship with kinematics and the resultant mechatronics will be taught slowly, and only when the example application makes it necessary. For example,

in this chapter we thought the students what a relay is and how it works in detail. Then we explained how they should be used to control a DC motor. This way a wide variety of students with different backgrounds will be able to benefit from this book and the Robot Vision course the book is used for.

4.6. Servo Motors

Figure 4.8 shows three of the analog servo motors among hundreds (if not thousands) available on the market. You can control a DC motor by only the voltage, the current and the duration of power you send to it. Although for many applications the movement it creates is precise enough, for many other applications such as robot arms and 3-D printers that is not enough. Servo motors allow the user to control how many degrees the motor rotates. That is a lot more precise than turning on a DC motor for "so many milliseconds" without exactly knowing how much load it is carrying and how much power left in the batteries.

Figure 4.8. Different types of servo motors.

There are many different types of servo motors. These motors differ in speed, torque and operating voltage and power. Some of the servos can turn limited degrees. For example, the servo on the right can turn 180 degrees at most. Many others can turn unlimited degrees meaning that you can turn them two full rotations if you command them to turn 720 degrees. Keep in mind that there are much bigger and powerful servos that look very different from these. The ones shown in Figure 4.8 are low power servos used by hobbyists.

Why are there so many low/medium power servos on the market? That is because depending on the application, the need for the servo's capabilities may change considerably. Therefore, one must be very careful when it comes to deciding about which servo to use (and where). The most important specification of the servo motors are their torque and their rotation angle. These two are the most decisive features for a servo. The third one is the rotation speed.

The torque of a low/medium range servo is between 1 to 50 kgf.cm (kilogram force centimeters). Meaning that a 5kg.cm torque servo can create a force of 5 kg when connected to a stick that is 1cm long. On the other hand, if you connect the servo to a 5cm stick as shown in Figure 4.9, then the force created at the tip of the stick drops down to 1kg. Considering the fact that a typical robot arm limb member varies between 20 to 30 cm (excluding toys), an average 25 cm robot arm limb could lift barely 200 g excluding the weight of the stick (limb) itself. Any useful robot arm would require five times the torque the servo in this example can generate.

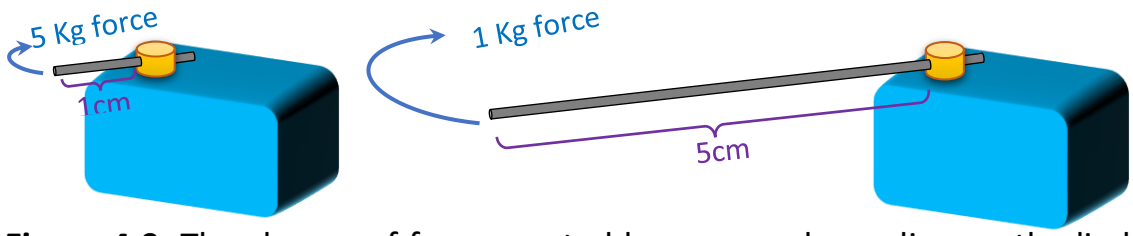


Figure 4.9. The change of force created by a servo depending on the limb length.

The next important spec of a servo is its rotation ability. If you want to use a servo for controlling the flaps of a model airplane, 90-degree servo might be enough for you. On the other hand, if you wish to use the servo as the joint of your robot arm, 90 degrees is usually not enough. Most joints need to rotate at least 120 degrees therefore you would need a 180-degree servo. If you want to use servo to rotate the wheels of a mobile robot, then you need a servo that can rotate several turns or a continuously rotatable servo.

Figure 4.10 shows several very common servos with different torques. They all have rotation angle of 180 degrees. The middle one (Tower Pro MG995) is the same one shown in Figure 4.8. If you remove the metal pin, you can use it as a 360-degree servo. Keep in mind that removing pins can cause the servos to

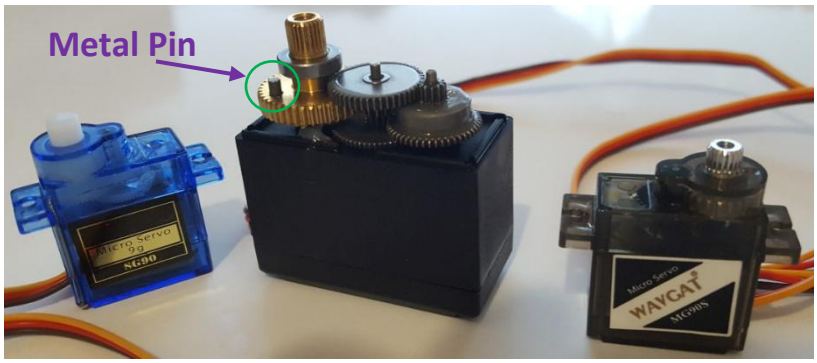


Figure 4.10. three different 180-degree servos. The one in the middle can be converted to 360-degree servo by removing the metal pin.

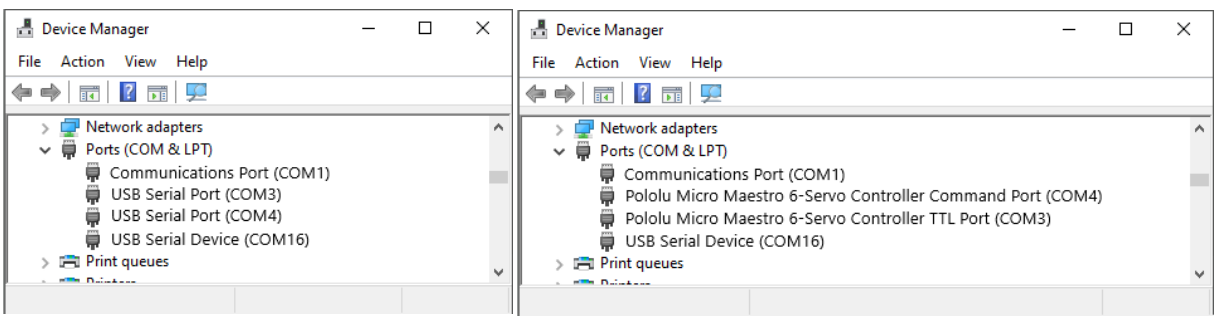
malfunction. The small ones cost 1.5 dollars and the large one costs 4 dollars (2025 prices). The blue one has a torque of 1.8 Kg.cm, the middle one has 15 Kg.cm and the one on the right has a torque of 2 Kg.cm. Very low torque servos have plastic (resin) gears and that is not a problem. However, you should avoid buying medium or high torque servos with plastic gears because they will not work well.

Some of the 360-degree servos can rotate continuously if you command it to turn beyond its limits. They may continuously rotate one way if you instruct it to rotate below zero and continuously rotate the other way if you instruct it to rotate beyond 360 degrees.

4.7. Controlling Servos

One of the most popular low-power USB servo controller manufacturers is Polulu. Their electronic cards called “Maestro” are available at any hobby/robot shop and can be easily purchased over the internet. They haven’t changed their design since the bronze age therefore, you can find many projects on the internet using their cards.

When you connect a Maestro card to the USB port of your PC, if your PC has Windows 11 or later version on it, Windows will create two serial ports in your system as shown in Figure 4.11-a even if you don’t install the Maestro device drivers. You can see those serial ports by opening the Device Manager (Control Panel→Device Manager) program. If you choose to install the Maestro device drivers, then the serial port names will change as shown in Figure 4.11-b. Either way, the I-See-Bytes library can easily interface with Polulu USB servo controllers. Here is how the interface works. First, we create an I-See-Bytes device:



(a) (b)

Figure 4.11. (a)Polulu Micro Maestro USB Card attaches itself usually to serial ports 3 & 4 even without installing the driver. (b) The name of the serial ports change after installing the driver.

```
ICDEVICE servo;
CreateMaestro(servo, 4);//Creates a servo control device using the
                        serial com port 4. Maestro card is connected
                        to this port and may control 6 to 24 servos.
SetTarget(servo,0,6000);//Tells servo card connected at comport 4 to
                        move the servo motor-0 to position 6000.
```

Keep in mind that if you have some other device using the serial port 4, and then you connect the Maestro card, it will choose some other port and you need to find out that port number in order to set the parameter of CreateMaestro(.).

Servos are controlled by parameters that use units of **µs** (micro seconds). That is because the servo controller sends a signal to the servo that stays "on" for so many milliseconds to instruct it to rotate. The neutral location (middle position) of every servo is at 1500 µs (1.5 milli seconds). The default resolution of Mastro cards is 4 times the resolution of a µs. Therefore, to make the servo move to its neutral position, we ask it to move 4×1500=6000. The default restrictions on the Maestro Servo card allows you to turn servos between 992 µs and 2000 µs (-45 and +45 degrees for most of the servos if they are not multiple rotation servos). In terms of SetTarget(.) function parameters, that translate to 3968 and 8000. If you try to set servo position beyond those limits it won't work. Pololu's explanation as to why they put those restrictions on servo channels is because moving them beyond these degrees may break the servo if it can't rotate beyond those degrees. In order to go beyond those degrees, you need to install the "Maestro Control Center" program on your PC. As shown in Figure 4.12, if you open the "Channel Settings" tab, you can remove those restrictions if you want to. In this figure, the limits of the channel 5 of has been altered to 64

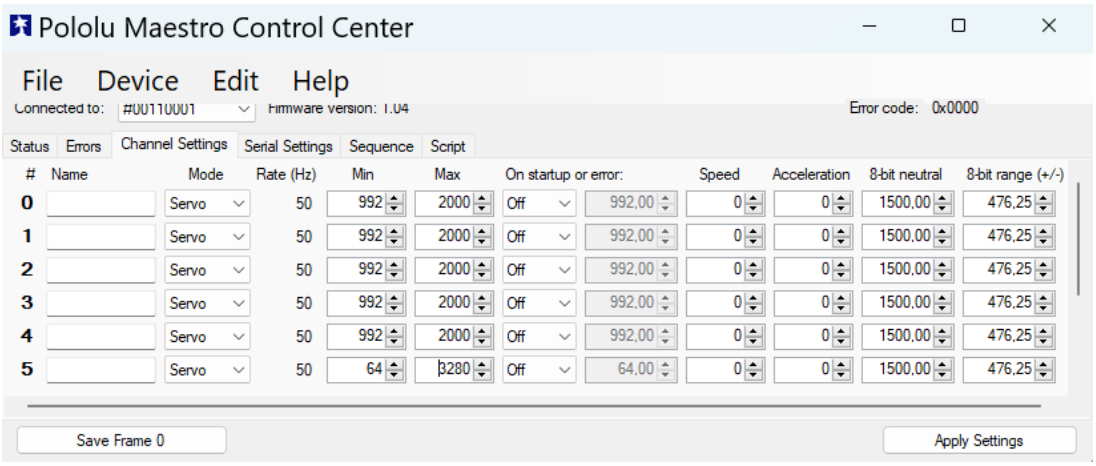


Figure 4.12. The "Channel Settings" tab on the Maestro Control Server software. You can alter the limits on the servo rotation angle of each channel using this software. Effects will be felt in the ICBYTES library immediately.

and 3280, which are the maximum limits for Maestro cards. When you press the "Apply Settings" button, these parameters get written on the card itself, not the program you see in Figure 4.12. Therefore, from that point on, that servo channel of that card will allow the SetTarget(.) function to set its rotation parameters between 256 (64 µs × 4 = 256) and 13120 (3280 µs × 4 = 13120) even if you disconnect that card from that PC and start using it in another PC. BE CAREFUL WHEN ALTERING WITH THOSE PARAMETERS. YOU MAY BREAK YOUR SERVO OR THE DEVICE CONNECTED TO YOUR SERVO.

Another important setting of the Maestro cards is in the "Serial Settings" tab of the Maestro Control Center program as shown in Figure 4.13. If you want to control the servos through the I-See-Bytes library, you must choose the "USB Chained" or "USB Dual Port" options and then press the "Apply Settings" button at the lower right corner. If you don't select one of those options, the ICBYTES library cannot command this card. Make sure that you do **NOT** accidentally choose the wrong setting.

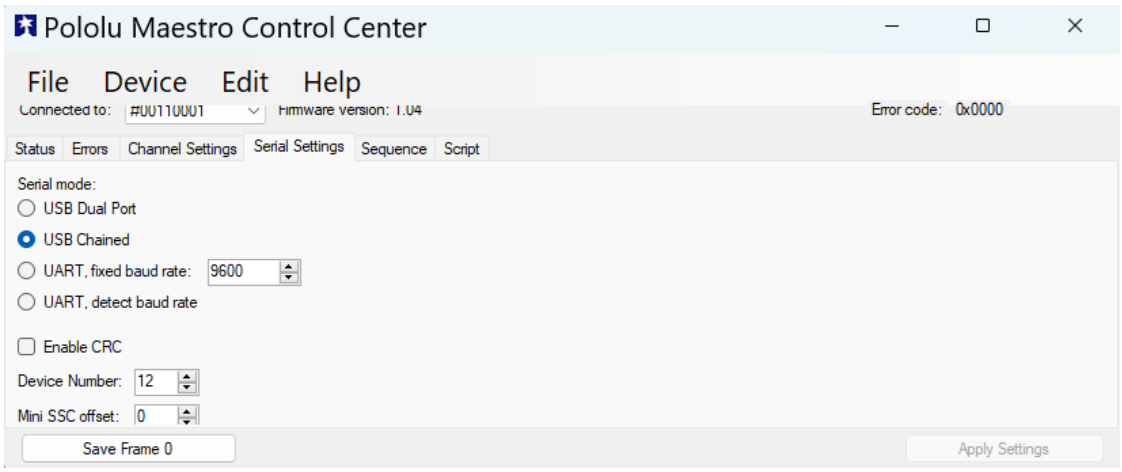


Figure 4.13. The "Serial Settings" tab on the Maestro Control Server software. If you want to be able to command servos using the I-See-Bytes library, you must select either "USB Chained" or "USB Dual Port".

Rest of the commands that you can use on the Maestro card are:

```
int t=GetTarget(servo,5);// It doesn't tell you the actual position of
                         the servo. This function tells you the what the
                         last position command for that channel was. It
                         is almost useless but included in the Maestro
                         protocol

WaitForServo(servo, 5); //Waits until the servo stops. If the load is
                         heavy or the servo is stuck for some other
                         reason it will return. Just because the servo
```

```
                                    stopped doesn't mean that it reached the degree
                                    you told it to (using Set Target(.) function).
```

Below is an example code that commands the servo at channel 5 to rotate -45 degrees, then wait for two seconds and then rotate to +45 degrees:

```
void CommandServo()
{
  ICDEVICE servo;
  unsigned short target = 1000 * 4;
  if (!CreateMaestro(servo, 4))return;
  ICG_printf("Setting ServoPos: %d ...", target);
  if(!SetServoPos(servo, 5, target))ICG_printf("Couldn't set target.\n");
  WaitForServo(servo, 5);
  ICG_printf("Current ServoPos: %d\n", GetServoPos(servo, 5));
  ICG_printf("Wait for 2 seconds ...\n"); Sleep(2000);
  target = 2000 * 4;
  ICG_printf("Setting ServoPos: %d ...", target);
  SetServoPos(servo, 5, target);
  if (!SetServoPos(servo, 5, target))ICG_printf("Couldn't set target.\n");
  WaitForServo(servo, 5);
  ICG_printf("Current ServoPos: %d\n", GetServoPos(servo, 5));
  CloseDevice(servo);
}
```

The output of this function on an edit window is shown below:

```
Setting ServoPos: 4000 ...Current ServoPos: 4000

Wait for 2 seconds ...

Setting ServoPos: 8000 ...Current ServoPos: 8000
```

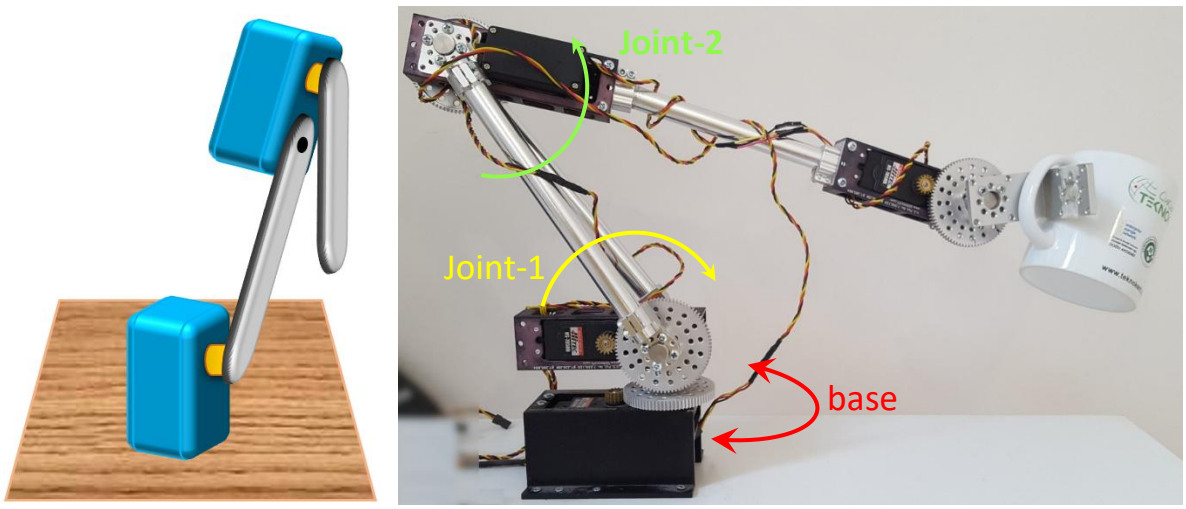
These commands are adequate for most of the servo control projects. However, in the future, more commands will be included in the I-See-Bytes library.

4.8. Building a Robot Arm

Building a robot arm with servos is very easy. First you need to fix the first servo on a table or a stiff plate using screws or glue. Then, using the plastic gadgets that comes with every servo, you can screw a metal, plastic or even wood stick to the servo's rotor. *Then fix another servo to the other end of the stick using glue or screws. Afterwards, again using the plastic gadgets that comes with the servos you bought fix another* stick to the rotor of the second servo as shown in Figure 4.14-a. You can continue this process until the bottom servo is no longer able to lift the rest of the robot arm.

Unfortunately, using the common servos available at robot shops, you cannot build a powerful robot that way. The highest torque servo you can find will be around 15 Kg.cm, which means that if your robot's reach is 40 cm, then the most weight it can lift will be 375 grams. That's not including the other servos and sticks you used. If you want to build a really powerful robot, then you need to use gears to further increase the torque.

Figure 4.14-b shows such a robot which is an old model of the Canadian robot company RobotShop. This arm not only uses powerful servos, it also employs gears to increase the torque fivefold. The servos it uses (Hitec HS-785HB) are designed to be used as sail winches on boats. The torque it provides is 13.6 Kg.cm when supplied with 6 volts. Amplified by the gears shown in the picture, each joint's torque reaches 13.6×5=68 Kg.cm. For a limb that is 20 cm, that gives you a pressure of 3.4 Kg at the tip of the limb. Such robots are quite powerful, and **can hurt the people around them!!!**



(a) (b)

Figure 4.14. (a) Simple robot arm. (b) A high-torque robot arm.

You may think that 3.4 Kg of pressure is not enough to hurt people. It is not powerful enough to hurt your bones however, if your flesh gets stuck between it and a hard place, you can easily bleed. Such a robot arm can easily drop breakable objects around them and they can lift themselves up and then cause themselves to drop on the floor and break. (Which happens to the author all the time. 😟)

When building such robots, or even when operating them, you need to make sure that the joints don't get forced beyond their limits. In order to ensure that, you need to manually rotate the rotors of the servos before fixing them to the arm joints so that the middle (neutral) position of the serve is the joint's

middle angle as well. This way you will ensure that not only you get the maximum rotation range for the limb but also a safer geometry that will avoid breaking the joint.

Furthermore, it is very important to decide a resting position for the robot arm. When the power is cut, the servos will lose their stiffness and the arm will bend under its own weight. The resting position must be a stable geometry such that the robot arm will not move or bend when the power is off. Another importance of this position is that you will need a predetermined position to move the robot arm when something unexpected happens, an error occurs or things go wrong. Always have an emergency power off switch near yourself. **The author accepts no responsibility for those who hurt themselves or cause financial damage**.

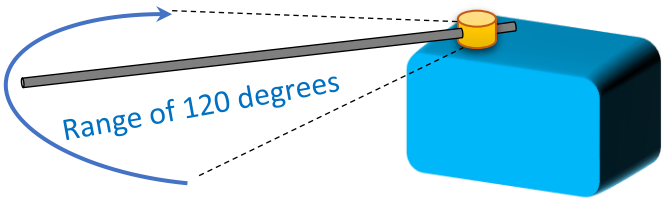


Figure 4.15. The servo must point the middle angle of its rotation range of the limb when it is at neutral state (1500 µs).

5. Device Access

As mentioned in the previous chapters, one of the most important features of the I-See-Bytes library is its built-in ability to interface with hardware devices. Any device, including virtual ones, can be accessed through the ICDEVICE object. Through this object, users can effortlessly interface with the common PC hardware such as the sound card, COMM ports, USB devices such as cameras and even low-level sectors of the disk drives. Some of these abilities were already listed in the book: “I-See-Bytes A Simplified C++ Library.” However, that book mentions only the abilities that the student version of the I-See-Bytes library provides. In order to be complete, they will be mentioned here as well. But we will also mention the hardware interfacing abilities of the Robot Vision (RV) version.

5.1. Querying Hardware Info

Before diving into hardware interfacing, we need to be able to query the hardware that is connected to our system including the system’s own hardware. The ICBYTES library has many functions that allows the user to do so.

5.1.1. System Information Query

Industrial quality applications need to know about the hardware of the computer they are running on so that they can utilize the hardware resources with optimum efficiency. For example, the amount of RAM, the type of processor or how many cores it has. The application shown below lists the following information about the system in order:

- Computer Name
- User name
- Number of processor cores
- Processor Type
- RAM (amount of memory)

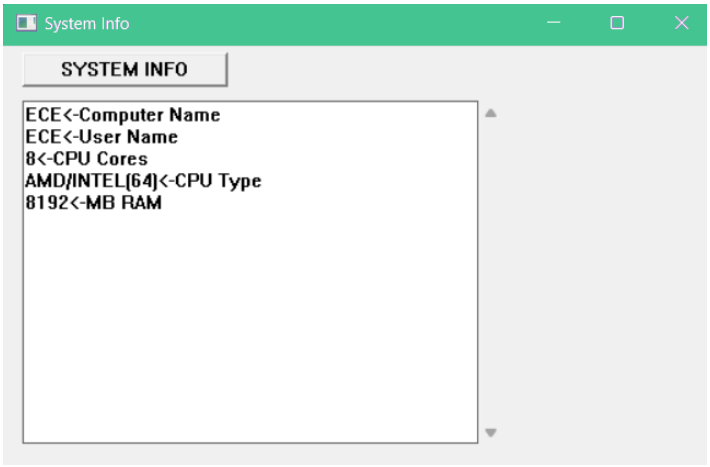


Figure 5.1. The screenshot of the application displaying the essential system info whose code is shown below.

```
#include"icb_gui.h"

int MLE;

void ICGUI_Create()
{
  ICG_MWTitle("System Info");
  ICG_MWSize(650, 450);
}

void Info()
{
  ICBYTES info;
  SystemInfo(info);
  Print(MLE,info);
}

void ICGUI_main()
{
  ICG_Button(15, 5, 150, 25, "SYSTEM INFO", Info);
  MLE = ICG_MLEdit(15, 40, 350, 250, "", SCROLLBAR_V);
}
```

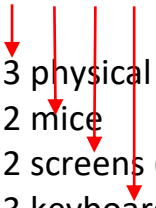
5.1.2. Querying Peripheral Devices

Devices that are connected to the motherboard are called peripheral devices. These devices include all of the USB devices that you connected or the ones that are inside the computer plus physical disk drives, special chips and sensors and even batteries. The commands we will use for this are:

```
ICBYTES info;
PeripheralInfo(info);
DisplayMatrix(MLE, info);
```

Those commands will display the following information on the multiline edit widget whose handle is MLE:

```
(SHORT 13 x 1)
3 2 2 3 1 0 0 9 0 2 1 0 0
```



3 physical disks (not logical)
2 mice
2 screens (monitors)
3 keyboards (including a virtual keyboard)
1 web camera
0 scanners
0 printers
9 usb port multipliers and devices
0 sensors
2 batteries
1 security (TPM) device

If this query tells us that there is a battery connected to our device, it means that we are currently using a laptop or a tablet. The fact that there are two screens indicates that there is another monitor connected to the laptop other than its own screen. There are also two physical keyboards meaning that the person who is using this laptop is not happy with the laptop's own keyboard so he/she connected a more ergonomic one through the USB.

5.1.3. Querying Logical Drives

In the previous section 5.1.2, we learned that the PeripheralInfo(.) function returns a matrix and the first element of that matrix is the number physical disks on your system. These are actual disk drive hardware that you can physically attach or detach to your computer. What if there are more than one partitions in your physical drives? For example, many users partition their disk and create C:\ and D:\ disks. They put all the programs in the C:\ disk and all of their data, songs, movies in the D:\ disk so that if one of them crashes, the other one stays recoverable. For a programmer, finding information about the logical disks is usually more important. The ICBYTES library has a function that queries the system and returns you a list of logical disk drives as a matrix:

```
ICBYTES drives;
if(GetLogicalDrives(drives))DisplayMatrix(MLE, drives);
else ICG_printf(MLE, "No Logical Drives or error.\n");
```

The output is an array of ones and zeros each representing the existence of a logical disk:

```
(UCHAR 32 x 1)
0 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A B C D E F G H I J K L M N O P Q R S ...
```

This computer has C:\, D:\, E:\, F:\ and H:\ logical disks available to the user.

5.1.4. Querying COM Ports

The PC has something called the COM ports. These are data pathways (that can be) connected to the CPU for serial communication. In the old days, all computers came with at least two RS-232 serial and one parallel port.

The Intel compatible CPUs have a thing called I/O ports. They should not be confused with the COM ports. I/O ports are more like resources that a CPU has while COM ports are actual communication hardware that use the CPU's I/O port resources. Usually, a serial COM port uses 8 I/O ports and an interrupt.

Any who, Figure 5.2 shows an actual RS-232 hardware that can be connected to the extension slots of your desktop computer. In the old days, when the USB was not invented yet, people connected peripherals through those ports. Nowadays, computers don't come with RS-232 or parallel ports because they are an outdated technology. USB is better, faster and more versatile and you can always purchase a USB to serial port converter if you need one.

However, many devices still use these serial ports internally. One of them is the Polulu servo controller card we mentioned in chapter 4. If you connect a Polulu Maestro servo controller card to your computer, and then open the "Device Manager" of your Windows operating system as shown in Figure 5.2,

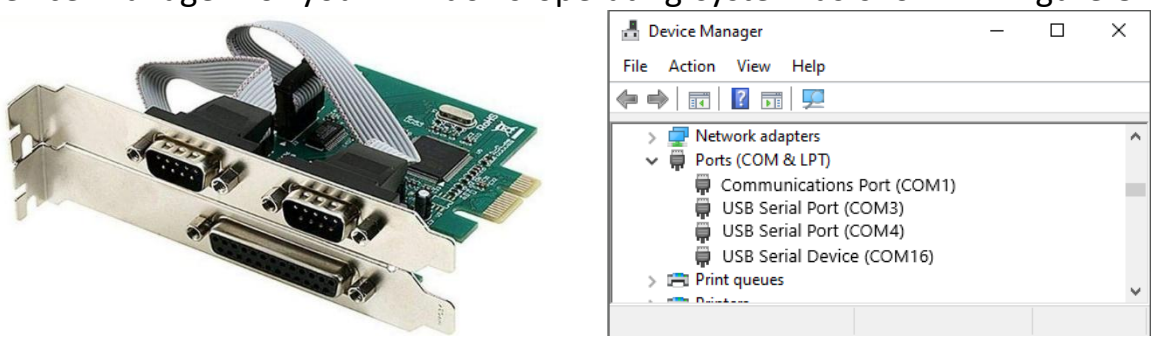


Figure 5.2. A PCI card (on the left) that provides two RS-232 serial and a parallel port to a desktop computer for those who miss the old days. How serial ports look in the "Device Manager" on the right.

you will see the serial ports it creates. The Polulu Maestro cards create two COM ports and they are usually on port 3 & 4 and labelled “USB Serial Port.”

On the other hand, another USB servo controller card brand, Lynxmotion, uses a USB UART device. UART stands for “Universal Asynchronous Receiver/Transmitter” and is another outdated technology. You need to install (if your computer doesn’t do that automatically) a UART→Serial driver on your computer in order to communicate with that device. When you install that driver, you will see that the Lynxmotion (SSC-32 card for example) will create a serial port at COM5 and label it as “USB Serial Device.” We shall learn about that device in more detail through the next sections of this chapter.

The point here is that despite being an outdated technology, the RS-232 serial port is being used by many Robotics related hardware as virtual devices because all major desktop operating systems have support for them. The more you work in this field, the more you will encounter motor controllers, sensors, LIDARS and even machine vision cameras that provide virtual or real serial COM ports as a means to transfer data or configuration parameters for that device. Therefore, it is vital for any RV library to provide the user with the ability to query COM ports. In the ICBYTES library, that is done easily as shown below:

```
ICBYTES ports;
if(COMPorts(ports)) DisplayMatrix(MLE, ports);
else ICG_printf(MLE,"No Ports or error.\n");
```

If you have COM ports established in your system as what the device manager shows in Figure 5.2, the you should get the following output:

```
(UINT 4 x 1)
16 4 3 1
```

Meaning that you have well-defined 4 COM ports at COM1, COM3, COM4 and COM16. You can access those numbers using “ports.U(.)”. The COMPorts(.) function returns true if there are at least one COM ports defined in your computer. Otherwise, it will return false and the matrix it returns will be empty.

5.2. Communicating with the COM Ports

The I-See-Bytes library have functions that allows you to communicate through the COM ports. But first of all, you need to learn the intrinsics of serial communication. Serial communication requires that you know exactly what

parameters that the device you want to communicate with is using. The most essential of those parameters is the "baud rate." Baud rate is the speed of communication and roughly translates to bits/second. The PC serial communications have fixed baud rates such as 1200, 1800, 2400, 4800, 7200, 9600, 14400, 19200, … and so on. You must set your baud rate to exactly that of the device you want to communicate with or your communication will be erroneous or not be established at all. Great majority of the robotic devices use 9600 baud because it is fast enough. The higher the baud rate, the shorter the communication cable has to be so it is a good trade off. (Never assume that it is always 9600! Read the specs of your device.) The function that opens / creates the serial COM port is:

```
ICDEVICE comport;
CreateCOMPort(comport, 4, 9600); // Establishes serial port COM device
                                    4 with a baud rate of 9600
```

There are many other parameters other than the baud rate which we will not discuss here because they are not the subject of Robot Vision. The reader must learn these concepts on her/his own.

In order to write or read data to the COM ports, we need an unsigned char (BYTE) type matrix. Here are the functions that read and write to a com port:

```
int WriteCOMPort(ICDEVICE& dev, ICBYTES& I);
int ReadCOMPort(ICDEVICE& dev, ICBYTES& M, short bytestoread);
```

You need to create the matrix yourself and put the data that you want to send to the device inside that matrix. To read data from the device, you needn't create a matrix, the function creates it for you. All you need to do is decide how many bytes you want to read.

Remember the Maestro servo controller card? We can communicate with that card by using the functions we learned in this section instead of using the Maestro card functions we mentioned in section 4.7. Here is a function that sets the servo targets to 4000 and 8000 just like the code example in that section:

```
void Serial()
{
      ICDEVICE comport;
      ICBYTES data;
      if(!CreateCOMPort(comport, 4, 9600))
      {
            ICG_printf(MLE, "Error: COM port not available!\n");
            return;
      }
      CreateMatrix(data, 4, ICB_UCHAR);
```

```
        int servopos = 4000;
        data.B(1) = 0x84; data.B(2) = 5;
        data.B(3) = servopos & 0x7f; data.B(4) = (servopos >> 7) & 0x7F;
        int byteswritten=WriteCOMPort(comport, data);
        ICG_printf(MLE, "Bytes written: %d\n", byteswritten);
        ICG_printf(MLE,"Wait for 2 seconds ...\n"); Sleep(2000);
        servopos = 8000;
        data.B(1) = 0x84; data.B(2) = 5;
        data.B(3) = servopos & 0x7f; data.B(4) = (servopos >> 7) & 0x7F;
        byteswritten = WriteCOMPort(comport, data);
        ICG_printf(MLE, "Bytes written: %d\n", byteswritten);
        CloseDevice(comport);
}
```

If your Maestro servo card is connected to your computer; if it shows up as serial ports 3 & 4 and a servo motor is connected to its 5th servo channel, then you should see the following output and also observe the servo's -45 and +45 degree rotations:

Bytes written: 4

Wait for 2 seconds ...

Bytes written: 4

5.3. Network Communication

The I-See-Bytes library facilitates network communications by presenting a simple interface for the user. Many users or students, especially students of engineering departments such as Electrical engineering, wouldn't know how to use network (Internet) communication. The I-See-Bytes library simplifies the TCP/IP communication through four simple functions:

```
bool CreateTCPServer(ICDEVICE& dev, int port);
bool CreateTCPClient(ICDEVICE& dev, const char* address, int port);
int RecvData(ICDEVICE& dev, ICBYTES& data);
int SendData(ICDEVICE& dev, ICBYTES& data);
```

The following codes demonstrate how these functions are used to communicate through the internet. Since we need two computers, one as client and one as server, to communicate with each other and not every student has two computers, instead we will establish a loop connection on a single PC. In other words, both the client and the server will reside on the same computer. The internet address "127.0.0.1" is a special address that always points to the computer itself. Therefore, if we ask our client program to connect to this address, it will connect to itself, but through the internet.

Since a server doesn't know when a client will connect to itself, it must continuously listen for a connection. This means that, the server code must be a parallel thread (a thread is a function that runs concurrently with our main function). Therefore, the following function must be called through a thread function. However, if you are not familiar with "Concurrent Programming" techniques, or creating threads, the I-See-Bytes library has a button exactly for that: The toggle button. The function that is called by the toggle button (see the book: I-See-Bytes A Simplified C++ Library) is a thread, meaning that the function called by this button runs parallel and concurrent to the main(.) function. All you need to do is create the toggle button as shown below inside the ICGUI_main() function:

```
ICG_TButton(900, 5, 120, 25, "START SERVER", Server, NULL);
```

Then, add the following function to your code:

```
void Server(void* p)
{
	ICDEVICE server;
	ICBYTES data;
	ICG_printf("Server Started.\n");
	while (1)
	{
		if (!CreateTCPServer(server, 27015)) break;
		int t = RecvData(server, data);
		if (t ==-1) {ICG_printf(MLE,"."); continue;}
		if (t == -3)break;
		ICG_printf(MLE,"Received % d bytes : \n", t);
		Print(MLE,data);
		CloseDevice(server);
	}
	ICG_printf("Server VLOSED!");
}
```

This code creates a server on your computer listening at the port 27015. Do not confuse these ports with ports of the serial COM devices. They are completely different. Internet ports allow you to create many different servers on the same computer. For example, if you have a mail server and a web server on the same computer, then they need to operate at different ports so that they can provide different services. Notice that, this code runs in an infinite while loop. That is because when a client connects and then quits, the server should be ready for a new client. The rest of the code waits to receive data and performs error checks. If everything is OK, it prints the number of bytes received from the client and also the message in those bytes on a multiline edit text box (widget).

The client code is even simpler. And you don't need a toggle button to call this function. An ordinary button is adequate and preferred. In the ICGUI_main() function, you add the following line to create a button that calls the client function:

```
ICG_Button(1020, 5, 150, 25, "Send Data", Client);
```

Then add the following code to your program:

```
void Client()
{
      ICDEVICE clnt;
      ICBYTES data;
      data = "Once upon a time ...";
      if (CreateTCPClient(clnt, "127.0.0.1", 27015))
      {
            int t=SendData(clnt, data);
            ICG_printf(MLE,"Sent % d bytes : \n", t);
      }
      CloseDevice(clnt);
}
```

This function opens a connection to the 27015th port of the internet address "127.0.0.1" which is the self-address of every machine. Then, if a successful connection is established, it sends the words: "Once upon a time ..." and prints the amounts of bytes it sent. When you run the program, first you need to press the "START SERVER" toggle button, so that the server starts first. Then you can press the "Send Data" button as many times as you like. Every time you press that button, you should see on the text box of your program the following lines:

Server Started.
Sent 20 bytes :
Received 20 bytes :
Once upon a time ...

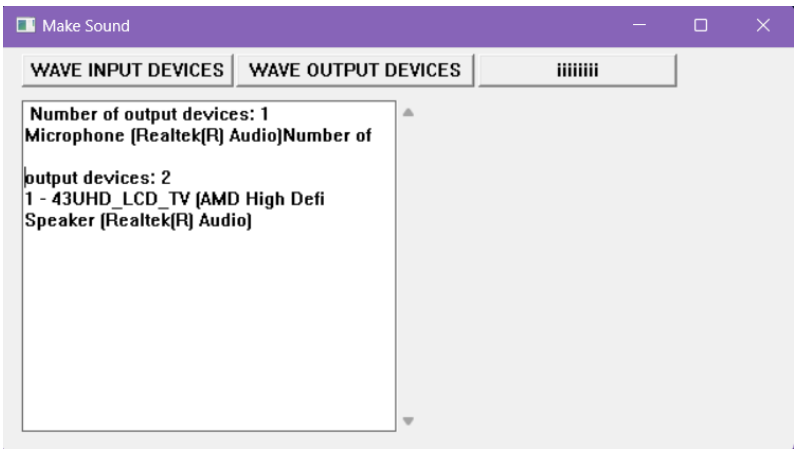
In this example we sent the contents of a "char" type matrix. You can create any type of matrix and send its contents, however, the matrix you receive using the RecvData(.) function will always be of type "unsigned char". In the future, more methods (functions) will be added to the ICBYTES library that will allow us to send exact replicas of matrices and vector lists. Keep in mind that you can use SendData(.) and RecvData(.) functions with both the server and client type devices.

5.4. Sound Output

The first step in device access is to query the number and properties of devices connected to the computer. However, unfortunately, many libraries cannot provide this feature. For example, the OpenCV library cannot even tell you the number of cameras installed on your computer. However, the I-See-Bytes library has very advanced system for polling devices compared to many other libraries. For example, the following commands not only tell you how many audio output devices you have, but also their brands and features:

```
ICBYTES output;
int number=WaveOutputDevices(output);
```

What is meant by “wave” here is a sound wave signal. However, since waves beyond the hearing threshold of the human ear can also be created, they are only called waves. This function expects a fluid in ICBYTES as input. Let's say, if there are two devices with audio output features connected to your computer's hardware, then it creates a 32x2 matrix in terms of characters and writes their brands and models on each line. In the application whose screenshot is given below, the number and names of the audio input and output devices connected to the computer it is currently running are shown.



When the first button from the left is pressed, the number and names of the wave input devices are written on the screen, and when the middle button is pressed, the number and names of the wave output devices are written. It is understood from this output that this computer's own speaker is connected to a Realtek sound chip. Again, the computer is most likely connected to a 43-inch UHD screen or television via HDMI cable and can also access its sound system. More importantly, the 43-inch display is currently set to be the default audio interface. The function called when the “WAVE OUTPUT DEVICES” button in the

middle is pressed is shown below. Here, “MLE” is the handle of the multi-line text box in which the device information is printed in the picture above. The entire code of this application is shown below.

```
#include"icb_gui.h"
int MLE;
unsigned char ii[67] ={139,131,127,108,108,96,67,84,54,31,40,18,9,11,5,11,12,
30,37,54,74,91,103,127,139,147,164,176,194,202,218,230,230,244,250,248,237,
226,225,201,182,171,139,117,98,74,56,37,26,24,9,9,11,16,24,37,43,58,75,84,84,
103,108, 122,124,138,141 };

void ICGUI_Create()
{
  ICG_MWTitle("Make Sound");
  ICG_MWSize(620, 350);
}

void InputDevices()
{
  ICBYTES input;
  ICG_printf(MLE, " Number of input devices: %d\n", WaveInputDevices(input));
  Print(MLE, input);
  ICG_printf(MLE, "\n");
}

void OutputDevices()
{
  ICBYTES output;
  ICG_printf(MLE, "Number of output devices: %d\n",WaveOutputDevices(output));
  Print(MLE, output);
  ICG_printf(MLE, "\n");
}
void iiiii()
{
  ICDEVICE d;
  ICBYTES i;
  if (CreateSound(i, 1, 3000, ICB_UCHAR, 8000) == 0)
  {
    for (int y = 0; y < 3000; y++) i.B(1, y) = ii[y % 61];
  }
  if (!CreateCompatibleDevice(d, ICB_WAVEOUT, 0, i))
    ICG_printf(MLE, "Can’t acccess sound card!\n");
  if(!WaveOut(d, i))
    ICG_printf(MLE, "Device cannot play this sound!\n");
  Sleep(400);
  WaveOut(d, i);
  CloseDevice(d);
}

void ICGUI_main()
{
  ICG_Button(15, 5, 160, 25, "WAVE INPUT DEVICES", InputDevices);
  ICG_Button(177, 5, 180, 25, " WAVE OUTPUT DEVICES ", OutputDevices);
  ICG_Button(360, 5, 150, 25, "iiiiiiii", iiiii);
  MLE = ICG_MLEdit(15, 40, 300, 250, "", SCROLLBAR_V);
}
```

As can be seen from the source code, this application's main purpose, apart from querying the audio input/output devices connected to the computer, is to produce the "iiii" sound. In order to do this, the wave information containing the digital data of this sound is written into the "unsigned char ii[67]" array at the beginning of the code. Then, when the button that makes the sound is pressed, the function called iiiiii() calls a function that we have encountered for the first time:

```
CreateSound(i, 1, 3000, ICB_UCHAR, 8000)
```

This function actually creates a matrix of size 1x3000 in bytes. However, since this matrix (vector) carries audio data, it creates some special data structures along with the matrix, just as the CreateImage() function to create a matrix that carries the image. Thus, when the user wants to listen to this sound, the process of playing the sound is faster because these special data structures are readily available. Additionally, the CreateSound() function expects an extra input, unlike the CreateMatrix() or CreateImage() functions. This is the last number "8000". This number sets the speed at which the audio data will be played. In other words, it indicates that the sound card will use 8000 samples of the data in this matrix every second. Since our matrix is 3000 long, this means that this matrix can only produce sound for 3/8 seconds = 375 milliseconds.

If you pay attention, the top array “ii[67]” is only 67 elements long. However, fortunately, the sound it produces is periodic, that is, a repeating sound. Therefore, if we write this index consisting of 67 elements repeatedly into our matrix (actually, it would be more accurate to call it our vector) consisting of 3000 elements, we can extend the duration of the sound "iiii". This is what the following for loop does.

Then we encounter a new function again:

```
ICDEVICE d;
ICBYTES i;
CreateCompatibleDevice(d, ICB_WAVEOUT, 0, i)
```

This function tells us that we want to reach the wave output device number zero and play the sound wave with numerical data in the "i" matrix on this device. What we need to remember here is that the i matrix contains mono, that is, single channel (if it were dual channel, it would be stereo), data in bytes and 8000 of these data must be played per second. Therefore, we need hardware that can do these. If the hardware is capable of playing this data, a data structure that provides access to this hardware is created in the "d" fluid and the hardware

becomes ready for use. In this case, the CreateCompatibleDevice() function returns true. If there is a positive return, all we need to do from now on is to call the WaveOut() function as many times as we want, send the "i" matrix to it and play the sound in the matrix. When we are done, we can delete the device access data created in the "d" fluid with the CloseDevice(d) command.

If we wanted to produce stereo sound, we would have to create our matrix with dimensions 2x3000 instead of 1x3000:

```
CreateSound(i, 2, 3000, ICB_UCHAR, 8000);
```

Then, we would have to write into i.B(2,y) along with i.B(1,y) inside the for loop.

5.5. Reading Sound Files

In the Windows operating system, the audio recording format called WAVE or WAV, which is the abbreviation of the words "Waveform Audio Format" developed jointly by IBM and Microsoft, is used. This format is one of the two most used formats, along with the AIFF format, especially for uncompressed audio files. All sound notifications made by the Windows operating system at startup, when you make an error or for any other reason are kept in ".wav" format files. You can access these files from the C:\Windows\Media file. To load audio files in this format into the matrix, all you have to do is call the ReadWave() function:

```
ICBYTES A;
ReadWave(A, "C:/Windows/Media/tada.wav");
```

If this function reads the file successfully, you can use another version of the WaveOut() function for convenience to play the sound matrix in fluid A:

```
WaveOut(A, 0);
```

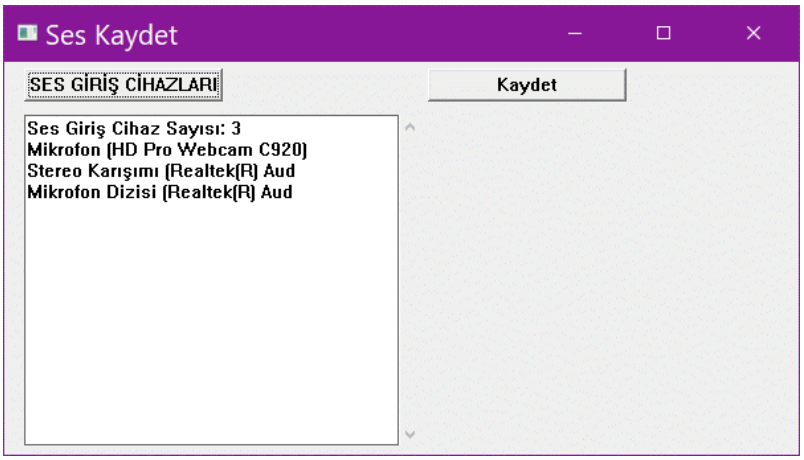
This function ensures that the audio data carried by matrix A in a single line is played by the fixed audio output device numbered zero. By the way, before playing the "tada.wav" file, if you execute the following command:

```
A*=6;
```

You will hear the volume increase considerably. Here, we increased the amplitude of the sound by multiplying the sound data by 6. If we multiplied by 15, you would hear distortion in the sound because we would have reached numbers that were too large (overflow) for the variable type used by the matrix.

5.6. Sound Recording

If we want to record sound from a microphone or another signal recording input into a matrix or vector, we need to repeat steps very similar to those we did in the previous section. First of all, we need to determine the device with which we will record sound or signal. For this, we must list the audio/signal input devices connected to our computer:



Unlike last time, this time we see three input devices connected to our computer. This time, we see that our zero numbered signal input device is the microphone of an HD PRO Webcam C920 model webcam from Logitech. The number one input device is a Stereo Mixer, and in the last row, number two, we see that the microphone input of our computer's sound card. The function shown below, which is called when we click the "Record" button, takes a recording from the webcam's microphone at a speed of 8000 samples per second for two seconds and then plays it back to us:

```
void Record()
{
  ICDEVICE d;
  ICBYTES i;
  CreateSound(i, 1, 16000, ICB_UCHAR, 8000);
  if(CreateCompatibleDevice(d, ICB_WAVEIN,0,i))
    ICG_printf(MLE,"Recording Started\n");
  WaveIn(d, i);
  ICG_printf(MLE, " Recording Complete\n");
  WaveOut(i, 0);
}
```

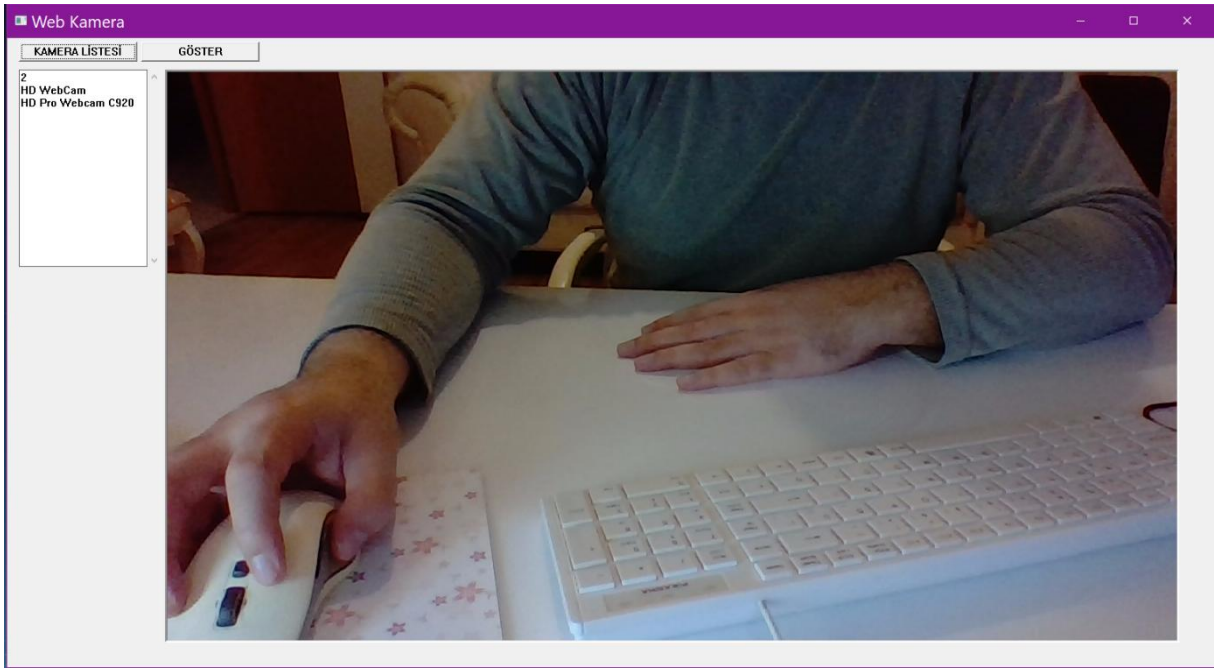
As explained in the previous section, we first created a vector using the CreateSound() function. Unlike last time, this time we created our vector with a length of 16000, not 3000. This is because we want to get a recording that will last two seconds. If we want the microphone to take 8000 digital samples per

second, we need a vector long enough to hold 16000 numbers for two seconds. Then we used the CreateCompatibleDevice() function again, but this time we gave the second parameter as “ICB_WAVEIN”. This parameter told the function that we wanted to create an input device. If you noticed in the previous section, this parameter was “ICB_WAVEOUT”.

Then, we placed the audio recording into the “i” vector by calling the WaveIn() function. Now there is nothing left to do but listen to the sound we recorded with the WaveOut() function. In this section, we learned how easy it is to access audio input and output with the I-See-Bytes library.

5.7. Camera Access

The application shown below shows the number of web cameras connected to our computer, model names and the image taken from the camera.



According to the text box, two cameras are connected to this computer. Of these, the fixture numbered zero is called "HD Webcam", which has a general name. Such names are often used to describe webcams that come on a notebook or tablet computer. From this description, we can guess that the computer being used is a laptop. We understand that the second camera is the HD PRO Webcam C920 model from Logitech. The source code of this application is shown below.

```
#include"icb_gui.h"

int MLE,FRM;

void ICGUI_Create()
{
  ICG_MWTitle("Web Kamera");
  ICG_MWSize(1550, 850);
}

void Kameralar()
{
  ICBYTES kamera_listesi;
  ICG_printf(MLE,"%d\n",GetVideoSourceList(kamera_listesi));
  Print(MLE, kamera_listesi);
}

void Goster()
{
  ICDEVICE d;
  ICBYTES i;
  CreateDXCam(d, 0);
  for (int x = 0; x < 100; x++)
  {
    CaptureImage(d, i);
    DisplayImage(FRM, i);
    Sleep(30);
  }
  CloseDevice(d);
}

void ICGUI_main()
{
  ICG_Button(15, 5, 150, 25, "CAMERA LIST", Kameralar);
  ICG_Button(170, 5, 150, 25, "SHOW", Goster);
  MLE = ICG_MLEdit(15, 40, 180, 250, "", SCROLLBAR_V);
  FRM = ICG_FrameDeep(200, 40, 1024, 1024);
}
```

In this application, we see that the function called GetVideoSourceList() is called when the “CAMERA LIST” button is pressed. This function takes an ICBYTES fluid as input. The function places a matrix of 32xcamera characters into this fluid and writes the name of one of the cameras connected to the system on each line of this matrix. Additionally, the function itself returns an integer, which tells us the number of cameras. Then, when we print the ICBYTES fluid (camera_list) that

we gave as input to the function, we see the model of the two cameras currently connected to the computer.

When we press the other button, SHOW, we see that the image from the camera is reflected on the screen for 3 seconds. When we examine the function that performs this, we encounter a new function used to create a device:

```
ICDEVICE d;
ICBYTES i;
CreateDXCam(d, 0);
```

This function creates an ICDEVICE type device structure that allows us to connect camera number zero. The term DXCam in the name of this function comes from the abbreviation of "DirectX Camera". What this means is:

If a camera is connected to your computer through any hardware channel, these routes may be of different standards such as USB, Ethernet, FireWire etc.; If this camera is introduced to the operating system via DirectX drivers, you can access it with the CreateDXCam() function.

So, in addition to USB cameras, you can also connect to network cameras if they have DirectX drivers. The student version of the I-See-Bytes library only provides access to cameras via DirectX. If you want to access IP, that is, network cameras via RTSP protocol, you should use higher versions of this library (for example, the Professional version).

Following the Show function, another command appears for the first time:

```
CaptureImage(d, i);
```

As the name suggests, this function captures an image from the “d” device, which in this case is the camera, into the “i” fluid. This picture is the image that the camera is currently looking at. Then this image is printed on the screen. When this process is repeated 100 times every 30 milliseconds, a 3-second live image is projected on the screen.

Microsoft has stopped using the Direct-X library. Although this library is still supported, it has recently started using a different library called “Microsoft Foundation Classes” (MFC). The I-See-Bytes library supports both libraries. If you want to access the camera using this library:

```
ICDEVICE mfcam;
if (CreateMFCam(mfcam, 0) < 0)return;
```

gives you access to the first camera. If you want to see what resolutions the camera supports:

```
ICBYTES modes;
if (ListCamModesTxt(mfcam, modes))
    Print(MLE, modes);
```

You can use the code. For example, for the camera on the laptop mentioned above, the following list will be displayed:

```
1280x720 30fps NV12
1280x720 30fps MJPG
640x480 30fps NV12
640x480 30fps MJPG
640x360 30fps NV12
640x360 30fps MJPG
1280x720 10fps YUY2
640x480 30fps YUY2
640x360 30fps YUY2
```

This list specifies the horizontal and vertical resolution (for example 1280x720), frame rate per second (fps) and image format. If desired, the user can select one of these resolutions when initiating the device. For example:

```
CreateMFCam(mfcam, 0,640,360);
```

With this command, you can specify that you want video in this specific resolution. If the user does not enter these parameters, the I-See-Bytes library will automatically select a suitable resolution from the top of the list. This is usually the highest resolution the camera can provide.

6. Edge Detection

6.1. Image Derivative

Before diving into edge detection, we need some mathematical background. We all know what the derivative is. It is the aim of a function at a given point. In order to calculate the derivative, we chose two x points that are Δ away from each other where the Δ→0. We calculate y=f(x) for those two points, and divide the height difference at those two points to Δ to obtain the derivative of that function. (Figure 5.1.) But we are dealing with digital signals that are discrete in nature. How do we calculate the derivative of a discrete function?

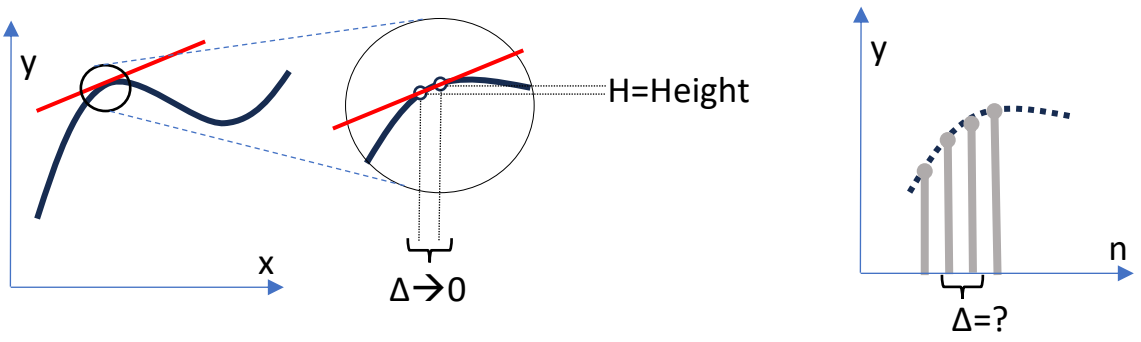


Figure 6.1. (a) A continuous function (signal) and infinitesimal delta (Δ) for derivative calculation. (b) Δ is not definable for a discrete function or signal.

If we write the equation for the derivative:

$$f'(x) = \frac{f(x) - f(x - \Delta)}{\Delta}$$

But for the discrete signal Δ is not defined. Therefore, we assume that Δ=1 and calculate our derivative as *f'(n)=f(n)-f(n-1)=y[n]-y[n-1]*.

Since the derivative for a discrete signal is the difference of two neighboring samples, it is called **the difference operator**:

y'[n]=y[n]-y[n-1]

For images, the signal is two dimensional. If we define the image signal as `I[x,y]` then the derivative can be along several directions. For example, if we want to calculate the derivative through purely horizontal direction,

```
I`[x]= I[x,y]-I[x-1,y]
```

And for purely vertical direction,

```
I`[x]= I[x,y]-I[x,y-1]
```

Or for both directions:

```
I`[x,y]= I[x,y]-I[x-1,y-1]
```

In mathematical terminology, two dimensional derivative is called a gradient and we shall do the same. The gradient along the x axis is described by the symbol G_x and the one in the y direction is called G_y. The magnitude of the gradient is:

$$G = \sqrt{G_x^2 + G_y^2}$$

The angle of the gradient is calculated using:

$$\theta = \text{atan}\left(\frac{G_y}{G_x}\right)$$

6.2. What is an Edge?

In an image, an edge is a border where a sudden change of contrast happens. How sudden? That is up to you. Consider the image shown in Figure 6.2. A white square (color 255) is drawn on top of a dark grey (100) background. If we draw these pixel values in 3-D as elevation, we get the yellow cube on top of the blue grid. That represents the sudden increase of the pixel values in the image. If we travel horizontally along the path of the black arrows, at the border of the white square we will encounter sudden change of contrast, as if we hit a wall. That is an edge. However, if we follow the path of the **red** arrow, at the bottom border of the cube, despite being on the border we will not experience an edge because we do not cross any borders.

This means that, if we scan image through the horizontal direction and calculate the gradient, we will discover only the left and right edges of the white cube. The top and bottom edges of the cube will be invisible in that gradient. We will prove this empirically in the next section.

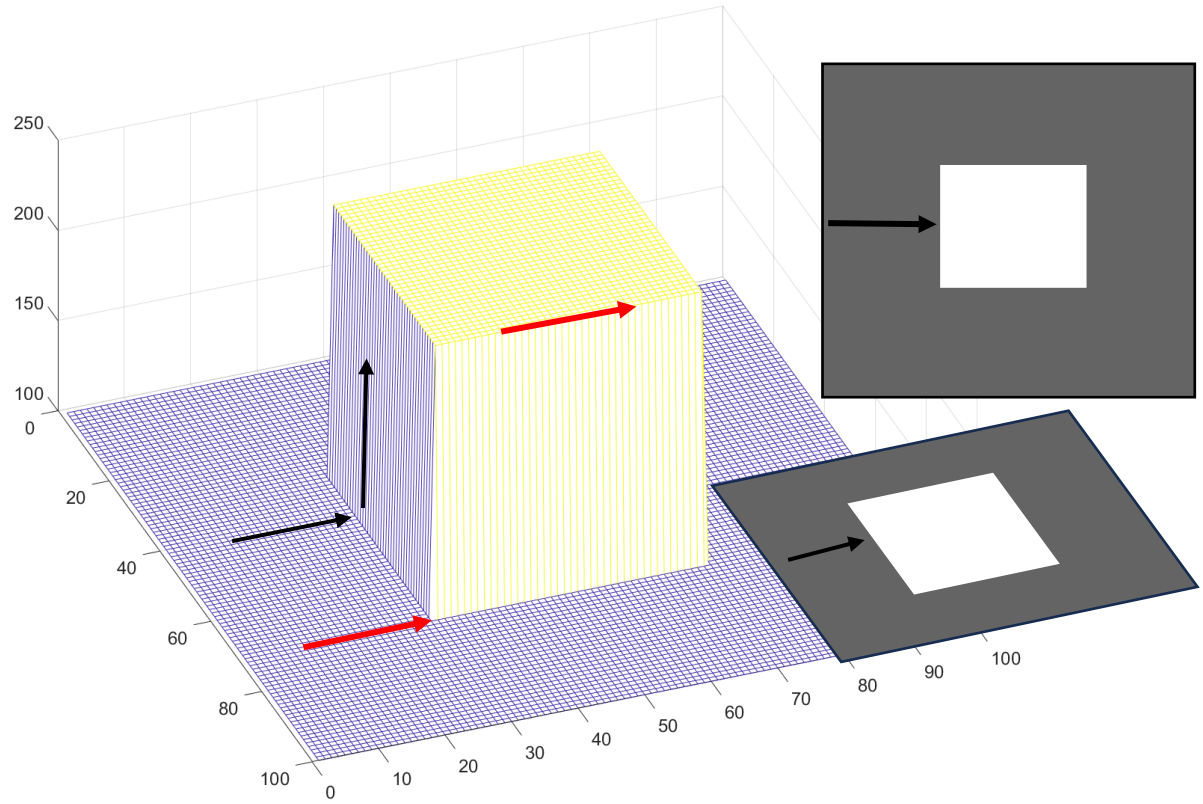


Figure 6.2. A white square drawn on a dark gray background and its 3-D elevation map using pixel values. A path through the dark arrow runs into the border where sudden change of contrast occurs, which is called an edge.

6.3. The Prewitt Operator

The Prewitt operator is a set of 2-D filter coefficients that are used to calculate G_x and G_y. As we mentioned before, the difference operator, which is the basic of gradient operators simply subtracts the adjacent values of a signal. Basically, it can be represented as {1, -1} as filter coefficients. However, we know that a filter must have odd number of coefficients so that the middle coefficient can be aligned on top of the target signal value. As a result of this the Prewitt operator is defined as: {1, 0, -1}. The problem with difference operators is that they are very sensitive to noise. Fortunately, we know how to reduce noise: A moving average filter. The simplest moving average filter is: {1, 1, 1} (remember we need odd number of coefficients.) If we calculate the cross product of these filters, it gives us the gradient operators in the x→ direction:

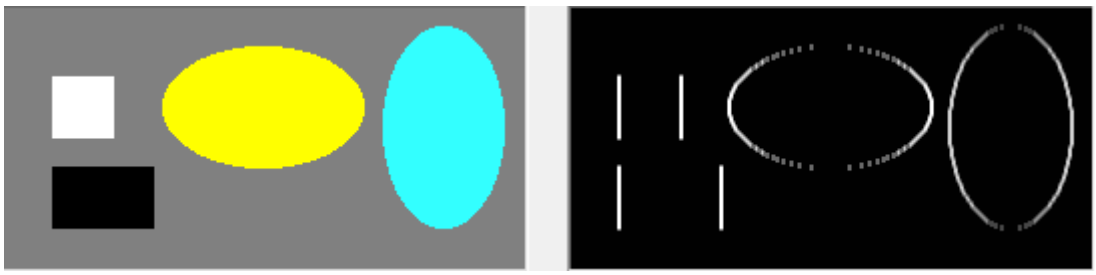
$$\begin{bmatrix}1\\1\\1\end{bmatrix} \times [1 \quad 0 \quad -1] = \begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}$$

and for the y↓ direction:

$$\begin{bmatrix}1\\0\\-1\end{bmatrix} \times [1 \quad 1 \quad 1] = \begin{bmatrix}1 & 1 & 1\\0 & 0 & 0\\-1 & -1 & -1\end{bmatrix}$$

Now consider the code below:

```
void GradientX()
{
	ICBYTES RGB,grey, outp;
	ICBYTES ver = { 1.0,1.0,1.0 };
	ICBYTES hor = { 1.0,0.0,-1.0 };
	CreateImage(RGB, 260, 130, ICB_UINT);
	RGB = 0x808080;
	FillRect(RGB,25, 35, 30, 30, 0xffffff);
	FillEllipse(RGB, 80, 20, 50,30, 0xffff00);
	FillEllipse(RGB, 190, 10, 30, 50, 0x33ffff);
	FillRect(RGB, 25, 80, 50, 30, 0);
	Luma(RGB, grey);
	Filter_V(grey, outp, ver, ICB_FLOAT);
	Filter_H(outp, grey, hor,ICB_FLOAT);
	abs(grey, outp);
	RoundFloat2Byte(outp,grey);
	DisplayImage(FRM1, RGB);
	DisplayImage(FRM2, grey);
}
```



(a) (b)

Figure 6.3. (a) Artificially created image with rectangles and ellipses. (b) Result of the horizontal Prewitt operator.

As seen from the Figure 6.3, the above function creates a 32-bit color image of size (260×130) and then makes each of the background components (RGB) 100 (RGB = 0x808080). Then it draws filled rectangles and ellipses on it using different colors. Since the filter only works on luminescence (8-bit greyscale) images, it calls the Luma(.) function. The output of this function is filtered in both directions with Prewitt coefficients. Because these coefficients create less than zero and greater than 255, the outputs of the filters are put inside float matrices. A derivative can create both positive and negative values. A rising edge will create positive while on the other hand a declining edge will create negative or vice versa depending the direction of your filters. Therefore, we cannot simply clamp negative values to zero. We need to take the absolute value of negative edges so that we can display them. To do that, we use the:

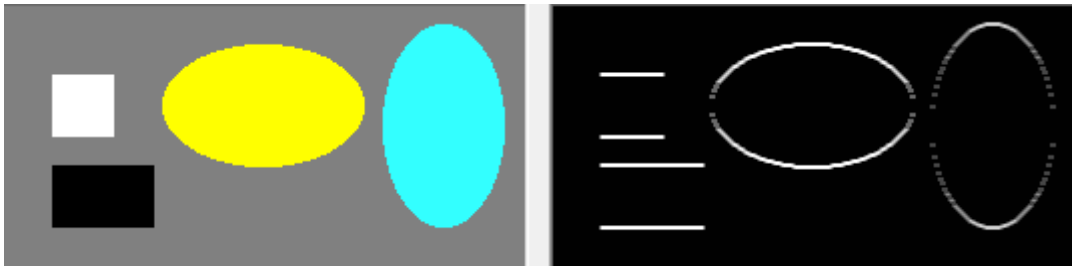
```
abs(grey, outp);
```

function. This function takes absolute value of each element in the "grey" matrix and puts the result at the corresponding place in the "outp" matrix. Afterwards we can round and limit the float values to byte (unsigned char) so that we can display the result as a greyscale image.

As expected, the horizontal filter detected only the horizontal edges (Figure 6.3-(b)) meaning that it shows only the left and right edges of objects. The top and bottom edges cannot be detected using the horizontal Prewitt operator. In order to detect those edges, all we have to do is switch the filter coefficients as shown below:

```
Filter_V(grey, outp, hor, ICB_FLOAT);
Filter_H(outp, grey, ver,ICB_FLOAT);
```

The result of this change is shown in Figure 5.4.



(a) (b)

Figure 6.4. (a) Artificially created image with rectangles and ellipses. (b) Result of the vertical Prewitt operator.

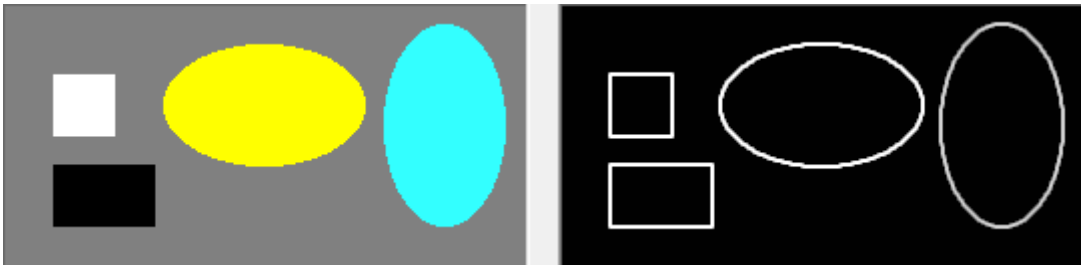
We have calculated G_x and G_y. Now we need to calculate the magnitude G using the formula given before:

$$G = \sqrt{G_x^2 + G_y^2}$$

The code that accomplishes that is shown below:

```
void GradMagnitude()
{
 ICBYTES RGB,grey, Gx,Gy,temp,outp;
 ICBYTES ver = { 1.0,1.0,1.0 };
 ICBYTES hor = { 1.0,0.0,-1.0 };
 CreateImage(RGB, 260, 130, ICB_UINT);
 RGB = 0x808080;
 FillRect(RGB,25, 35, 30, 30, 0xffffff);
 FillEllipse(RGB, 80, 20, 50,30, 0xffff00);
 FillEllipse(RGB, 190, 10, 30, 50, 0x33ffff);
 FillRect(RGB, 25, 80, 50, 30, 0);
 Luma(RGB, grey);
 Filter_V(grey, temp, ver, ICB_FLOAT);//Horizontal
 Filter_H(temp, Gx, hor,ICB_FLOAT);//Horizontal in GX now
 Filter_V(grey, temp, hor, ICB_FLOAT);//Vertical
 Filter_H(temp, Gy, ver, ICB_FLOAT);//Vertical in Gy now
 square(Gx, temp); Gx = temp; //square Gx
 square(Gy, temp);//square Gy
 temp += Gx;//temp has Gx^2 + Gy^2
 sqrt(temp, outp, ICB_FLOAT);// outp has square root of temp
 RoundFloat2Byte(outp,grey,true);//Round temp to byte-->grey(image matrix=true)
 DisplayImage(FRM1, RGB);
 DisplayImage(FRM2, grey);
}
```

The result of this function is shown in Figure 5.4.



(a) (b)

Figure 6.5. (a) Artificially created image with rectangles and ellipses. (b) Result of the magnitude of the image gradient.

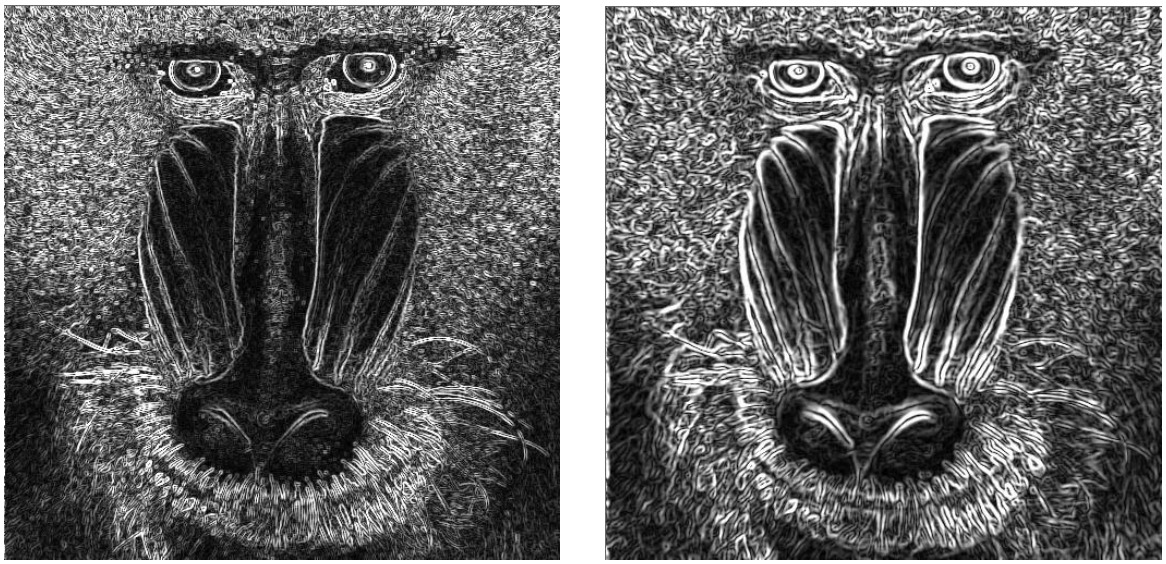
The crisp edges you see on Figure 6.4 might make you think that we have successfully constructed an edge detector. You would be wrong. A good edge detector has many more steps as you will see in the next sessions. For now, let's do the following exercise:

EXERCISE 6.1: Apply the code that generates the gradient magnitude image on the mandrill picture. (Result is shown in Figure 5.6.a.)

EXERCISE 6.2: First low-pass filter the mandrill picture and then apply the code that generates the gradient magnitude image. Hint: multiply G with 3 before rounding to get a brighter image. Use filter coefficients:

```
ICBYTES filter = { 0.05,0.1,0.2,0.3,0.2,0.1,0.05 };
```

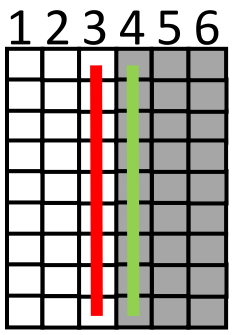
(Result is shown in Figure 5.6.b.)



(a) (b)

Figure 6.6. (a) Answer to the Exercise 6.1. (b) Answer to the Exercise 6.2.

6.4. A Philosophical Question



Consider the figure on the left. It shows the magnified pixels of an image where a white area is adjacent to a dark area. Where should we draw the edge? Should we draw it through the 3rd column marked with the red line or should we draw it through the 4th column marked with the green line?

We can clearly see that the edge is between the 3rd and the 4th columns but unfortunately there is no 3.5th column in the picture. In that case we need to make a decision. If we chose the 3rd column (the red line), this means that we consider that the dark area is the background and the bright area is the foreground object. On the other hand, if we choose the 4th column (the green line) this means that the dark area is the object and the bright area is background. Ideally an edge detector should allow the user to decide which is which. Unfortunately, the author of this book has never encountered an implementation that gives the user that option. Therefore, we should be aware of this shortcoming of the edge detectors if our algorithms depend on the precise locations of the edges.

6.5. Thresholding

If you carefully look at the images in Figure 6.6, you will notice that the edges are different than those in Figure 6.5.b. The edges in 6.5.b are very crisp, bright and uniform. On the other hand, some of the edges in 6.6 are dim and vary in thickness especially in Figure 6.6.b. That is because the edges of artificial objects such as rectangles and ellipses are well defined and the contrast differences they create is sudden. Unfortunately, an image of a real object does not have such well defined contours most of the time nor such sudden contrast change. Therefore, their edges can be dim and thick. What we want is fully bright and 1-pixel wide edges. We can accomplish that by thresholding.

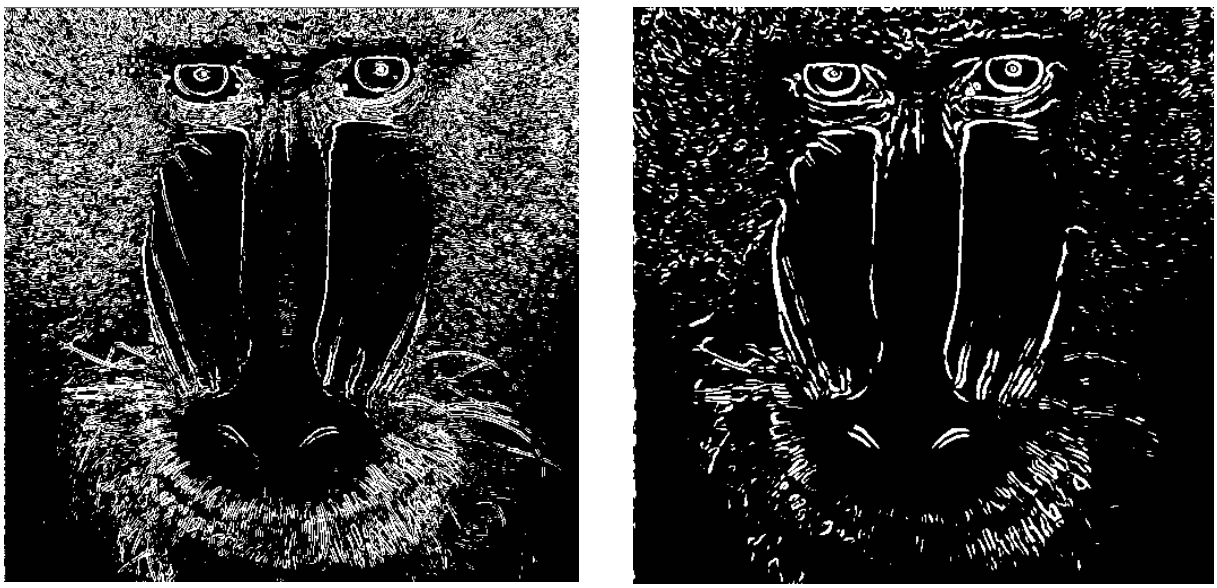
I-See-Bytes library has a function that does exactly that and it is called GreyThresh(.). In order to be able to call that function you need to include the "ic_image.h" header file:

```
#include"ic_image.h"
...
......
RoundFloat2Byte(outp, grey);
GrayThresh(grey, outp, 70); //GrayThresh(inp,outp,threshold);
```

As the name implies, GrayThresh(.) thresholds the input matrix. The input matrix must be a grayscale image or in other words a matrix of type unsigned char. The final parameter, which is 70 in our example is the threshold. This function scans the input matrix and if an element's value is less than the threshold value, then in the output matrix the corresponding element is set as zero and 255 otherwise. The output is also a grayscale matrix/image. Using this function, if we threshold the images in Figure 6.6.a and 6.6.b, we get the images in Figure 6.7.a and 6.7.b respectively. We use the threshold value 130 for the first image (a) and 70 for the second image (b). Our goal is to extract as much of the nose and eyes while as less of the hair as possible. Remember not to multiply Figure 5.6.b by three as recommended by the exercise because we don't need that anymore.

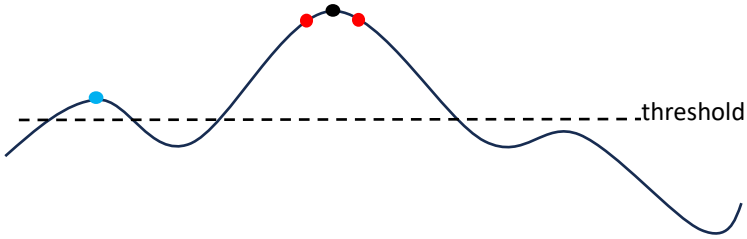
Figure 6.7.b clearly demonstrates that low pass filtering reduces the unwanted noise and texture. While the structural objects, the nose and the eyes are clearly observable, the entangled hair has disappeared. Unfortunately, the lines around the nose and the eyes have got thicker as well. That is because the low pass filter smeared the edges to bigger area. After edge detection, we prefer the edges to have single pixel thickness. One way to do that is to look for the maximum points of the gradient *G*.

$$G = \sqrt{G_x^2 + G_y^2}$$



(a) (b)

Figure 6.7. (a) Image in Figure 3.6.a after threshold at 130. (b) Image in Figure 6.6.b after threshold at 70.



So how do we figure out the maximum point of the gradient? Consider the curve in Figure 5.8. The black dot at the peak of the curve is greater than both of its neighbors (red dots). Therefore, we can say that black dot is the maximum of this curve. We can do this on a 3-D surface as well, if a point is greater than its 8 neighbors, then it's a local maximum point. We cannot accept all local maximums; the value of that maximum should be greater than threshold otherwise we will start including noise and background texture as edges. Only the peaks that are greater than the threshold should be accepted.

How do we determine the threshold? Some algorithms let the user chose the threshold while others set or recommend a preset value. Yet others try to determine the optimum threshold automatically. The automatic thresholding is a long discussion which we will study further in the next chapters. The most famous edge detector, the Canny edge detector uses double threshold. The large threshold is exactly like the threshold line we used in Figure 6.7. The pixels detected using the lower threshold are accepted as edges only if they are connected to edge points detected using the large threshold. In the ICBYTES library, the Canny edge detector is employed using the following function:

```
void Canny(ICBYTES& inp, ICBYTES& outp, float lowThrsh, float hghThrsh)
```

Below is the output when applied to the Mandrill image after low pass filtering.

7. Spectral Transform

7.1. The Fourier Transform

Remember how we tried to separate a signal composed of two different sine signals in the second chapter? We used a moving average filter that is exactly the same length as the first (short period) component to cancel it out which gave us the second (long period) signal. We used the knowledge that if you integrate a sine signal over one period you get zero.

A mathematician called Joseph Fourier discovered that if you multiply a continuous (infinitely periodic) sine wave with another one at every point and then sum the result (integrate the multiplication) then the result is zero unless those two sine waves have exactly the same frequency (period). This means that, if you take any periodic signal, such as the one when we say “aaaaa..”, as shown in Figure 7.1, and then multiply and integrate with different frequency sine or cosine signals, then you can determine its frequency components. Or in other words, you can construct that periodic “aaaaa..” signal by combining bunch of pure sine and cosine signals.

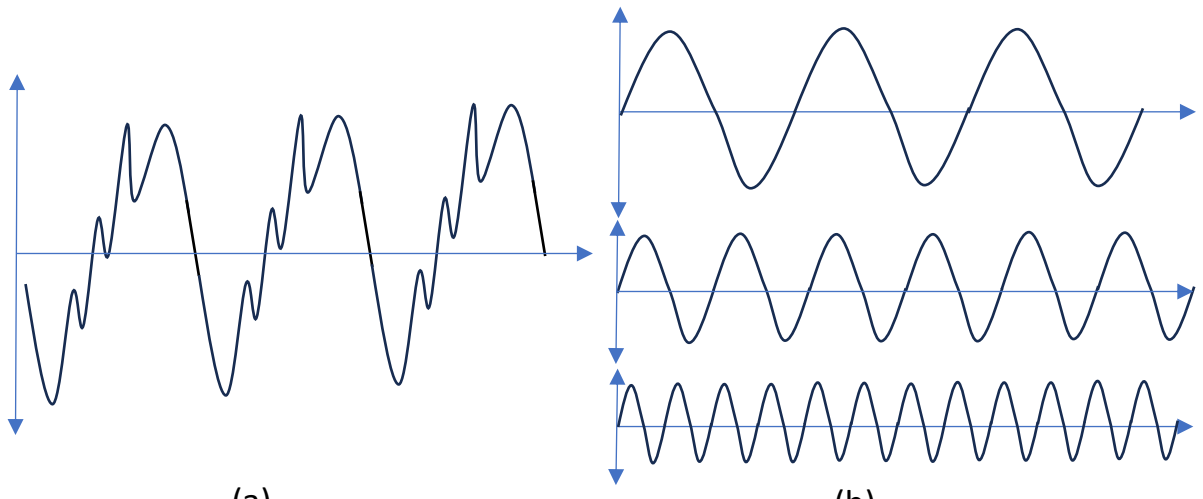


Figure 7.1. (a) The waveform of the sound “aaaa…”. **(b)** Its spectral components.

Decomposing a signal to its sine and cosine components may become quite handy if you want to analyze or modify that signal. You can increase or suppress its components as if a filter does and then reconsrtuct the signal from its components to have the desired filtering effect. Therefore, in image processing, one way to use the Fourier transform is to filter images. Unfortunately, for many image processing applications, Fourier transform is computationally very expensive. Other then special areas, it is not widely used. As a matter of fact, depite having spent dozens of years both in the industrial R&D and the academic reasearch, the author of this book have not had to use this method more than several times. Therefore, the Fourier transform will not be explained any further. We will spend our precious time learning its sister, the Discrete Cosine Transform.

7.2. The Discrete Cosine Transform

Just like in the Fourier transform, a discrete signal can be decomposed into cosine waves with different frequencies. The method for doing that was proposed by Nasir Ahmed in 1972. The advantage of using the Discrete Cosine Transform (DCT) is that it generates only real numbers as opposed to the Fourier Transform which produces complex numbers. If your signal is in x[n] with a length of N samples, then its DCT is expressed by the following formula:

$$D[y] = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} \left(x[n] \cdot a_y \cdot cos \frac{y\pi(2n+1)}{2N} \right)$$

$$a_y = \begin{cases} \frac{1}{\sqrt{2}}, & y = 0 \\ 1, & y > 0 \end{cases}$$

For this formula, the a_y coefficient is 1 for y=0, and $1/\sqrt{2}$ for y>0. That goes for the inverse of this transform as well:

$$x[n] = \sqrt{\frac{2}{N}} \sum_{y=0}^{N-1} \left(D[y] \cdot a_y \cdot cos \frac{y\pi(2n+1)}{2N} \right)$$

Note that the index dependency is different for the a_y in the inverse transform making it slightly different.

The main reason why we use the DCT is because of its compression ability. Consider the following code:

```
#include"icb_gui.h"
#include"ic_image.h"

int MLE, FRM1, FRM2, FRM3;

void ICGUI_Create()
{
	ICG_MWTitle("DCT TEST");
	ICG_MWSize(1800, 1000);
}
void DCT_Horizontal();
void DCT_Vertical();
void DCT_IDCT();
void Precision();
void DCTEight();

void ICGUI_main()
{
	ICG_SetFont(30, 0, "Calibri");
	ICG_Button(5, 5, 250, 25, "HORIZONTAL DCT", DCT_Horizontal);
	ICG_Button(260, 5, 280, 25, "VERTICAL DCT", DCT_Vertical);
	ICG_Button(550, 5, 280, 25, "DCT and then IDCT", DCT_IDCT);
	ICG_Button(830, 5, 280, 25, "PRECISION LOSS", Precision);
	ICG_Button(1120, 5, 280, 25, "DCT8x8 and IDCT8x8", DCTEight);
	FRM1 = ICG_FrameSunken(5, 80, 512, 512);
	FRM2 = ICG_FrameSunken(600, 80, 300, 300);
	MLE = ICG_MLEditSunken(1170, 80, 600, 510, "", 1);
	FRM3 = ICG_FrameSunken(5, 620, 1700, 300);
}

void DCT_Horizontal()
{
	ICBYTES RGB, grey,dct,outp;
	ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
	DisplayImage(FRM1, grey);
	DCT_H(grey, dct);
	dct /= 2.0;
	RoundDouble2Byte(dct, outp, true);
	DisplayImage(FRM2, outp);
}
```

This program is very similar to the one we used for filter demonstrations in chapter 2. Therefore, the screenshot of it will not be shown here. The function that is called by the first button, labeled “HORIZONTAL DCT” is also shown here. Other button functions will be shown in the following pages. This function loads the “Mandrill” image, converts to grayscale, and then calls a function called DCT_H(.). This function treats every row of the matrix/image “grey” as a signal and computes their DCTs and puts the results in corresponding rows of the (double) matrix “dct.” We divide every element of this matrix by 2 and then convert it to a grayscale image by using RoundDouble2Byte(.) function and

display it on the screen. The following function does the same thing in the vertical direction, which is called when we press the “VERTICAL DCT” function:

```
void DCT_Vertical()
{
	ICBYTES RGB, grey, dct, outp;

	ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
	DisplayImage(FRM1, grey);
	DCT_V(grey, dct);
	dct /= 2.0;
	RoundDouble2Byte(dct, outp, true);
	DisplayImage(FRM2, outp);
}
```

Figure 7.2 shows the results of these functions. Keep in mind that depending on the speed of your computer, DCT calculations may take a few seconds or more. When we look at the DCT performed in the horizontal direction, we see that the value of the DCT decreases towards the right →. Since this is a greyscale image, the darker the pixels, the lower their values. On the other hand, when we look at the DCT that was calculated in the vertical direction, we see that the lower pixels are darker. These both images tell us that the DCT tends to accumulate the signal information in the first few dozen samples of its output rather than the last few dozens. In other words, for DCT, the first output samples are the most important.

It is this property of the DCT that image compression algorithms take advantage of. If the later output samples of the DCT transform tend to get smaller, then we can set them to zero without losing significant amount of information. In next sections we will do exactly that. However, before doing that we need to address another important topic.

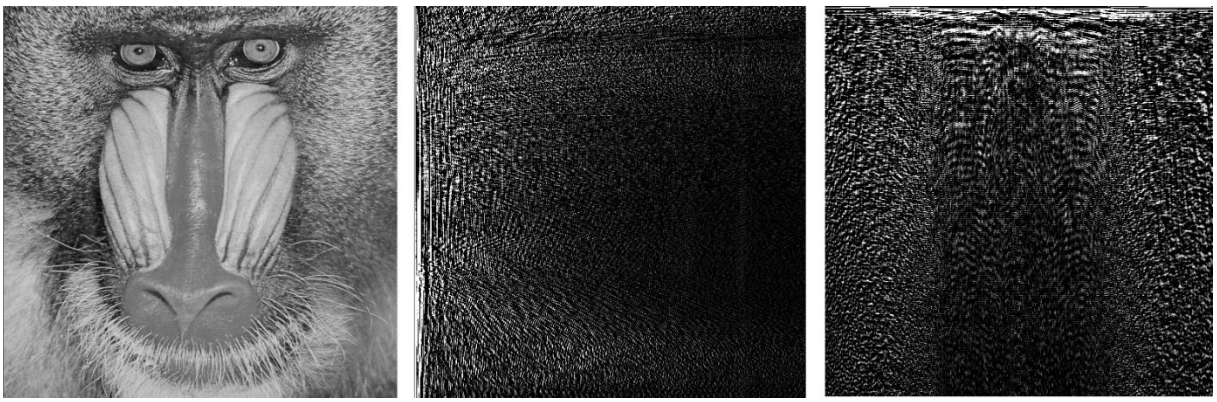


Figure 7.2. The DCT of the grayscale Mandrill image in horizontal and vertical directions.

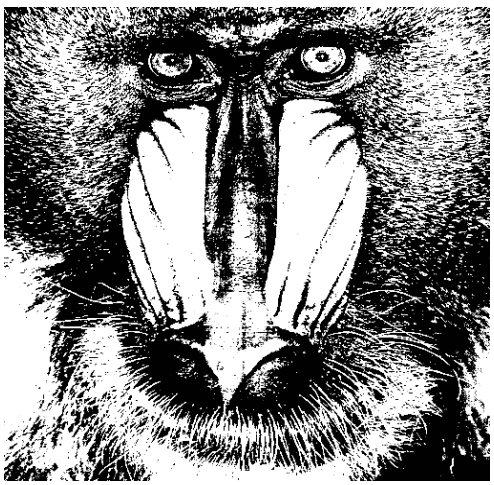
7.3. Precision Loss

As computer programmers, we all know that floating values cannot always be recorded in digital environments precisely. That is why we have the “float” and the “double” variable types in C++. Float is 32 bits long and despite the fact that it was precise enough for many calculations, eventually it was discovered that for scientific calculations a more precise variable type was needed hence the 64 -bit long “double” was introduced to the C++ language. At the beginning of the 1990s, it was understood that even “double” type is not precise enough for some calculations and the C++ community introduced the 128-bit long “long double” variable type. Unfortunately, at the time of writing, the Microsoft Visual Studio compiler does not support the long double type. Since the I-See-Bytes library was written using the MS Visual Studio compiler, we were not able to benefit from such high precision.

As the name implies, the DCT depends on cosine calculations which are numbers between -1 and +1. These can be very small numbers and since many of them are multiplied with pixel values and summed, the precision loss can be considerable. We can demonstrate that by taking the DCT and the inverse DCT (IDCT) of the Mandrill image and see how much information we lose. The following code does exactly that. Note that, in order to truly calculate the DCT of a 2-D signal, which is a digital image in our case, you need to first take DCT in one direction (say horizontal) and then take DCT in other direction (say vertical) **of the output of the first (horizontal) DCT**.

The output of this function is shown below. The loss of color depth of the mandrill image after a full two-dimensional DCT and inverse DCT demonstrates the consequences of not using floating point variables with enough precision.

```
void DCT_IDCT()
{
  ICBYTES RGB, grey, dcth, dctv, outp;
  ReadImage("mandrill.bmp", RGB);
  Luma(RGB, grey);
  DisplayImage(FRM1, grey);
  DCT_H(grey, dcth);
  DCT_V(dcth, dctv);
  IDCT_V(dctv, dcth);
  IDCT_H(dcth, outp);
  RoundDouble2Byte(outp, grey, true);
  DisplayImage(FRM2, grey);
}
```



To better understand the loss of precision, lets run the following codes in I-See-Bytes and Matlab. They both create the same matrix shown below and then calculate the vertical DCT. But the results are slightly different.

```
void Precision()
{
  ICBYTES inp, outp;
  CreateMatrix(inp, 3,10, ICB_DOUBLE);
  for (int y = 1; y < 11; y++)
  {
    Set(1, y, inp, y - 1);
    Set(2, y, inp, y+7);
    Set(3, y, inp, 24-y);
  }
  inp *= 0.1234;
  DisplayMatrix(MLE, inp);
  DCT_V(inp, outp);
  DisplayMatrix(MLE, outp);
  IDCT_V(outp, inp);
  DisplayMatrix(MLE, inp);
}
```

```
for(y=1:10)
  a(y,1)=y-1;
  a(y,2)=y+7;
  a(y,3)=24-y;
end
a=a*0.1234;
dct(a)
```

```
0.0000 0.9872 2.8382
0.1234 1.1106 2.7148
0.2468 1.2340 2.5914
0.3702 1.3574 2.4680
0.4936 1.4808 2.3446
0.6170 1.6042 2.2212
0.7404 1.7276 2.0978
0.8638 1.8510 1.9744
0.9872 1.9744 1.8510
1.1106 2.0978 1.7276
```

As we see below, the first rows (**green**) of the matrices created by the I-See-Bytes and the Matlab are the same. But the rest of the rows are slightly different. That's because the Matlab program uses higher precision arithmetic. Any signal that is longer than 8 samples may become corrupt when DC transformed using the 64-bit "double" variable. These are very small precision losses, less than 0.14 for the worst case, and probably not enough to alter pixel values more than 1 level. Nevertheless, they are there so be aware. When we apply the IDCT back to the DCT values we calculated with ICBYTES, we see even worse corruption. Compare those values to the original matrix input above↑. The Mandrill image is 512×512 pixels and the corruption for a 512-sample long signal is considerable. That's why the compression algorithms do not calculate the DCT for the entire image.

RESULTS OF I-SEE-BYTES:		
1.7560	4.8778	7.2192
-1.2451	-1.2451	1.2451
-0.0000	-0.0000	-0.0000
-0.1334	-0.1334	0.1334
-0.0000	-0.0000	-0.0000
-0.0436	-0.0436	0.0436
0.0000	0.0000	0.0000
-0.0176	-0.0176	0.0176
-0.0000	-0.0000	-0.0000
-0.0049	-0.0049	0.0049

RESULTS OF MATLAB:		
1.7560	4.8778	7.2192
-1.1137	-1.1137	1.1137
0.0000	0	0.0000
-0.1193	-0.1193	0.1193
0.0000	0	0.0000
-0.0390	-0.0390	0.0390
-0.0000	0	0.0000
-0.0158	-0.0158	0.0158
0.0000	0	0.0000
-0.0044	-0.0044	0.0044

IDCT with I-SEE-BYTES		
-0.0655	0.9217	2.9037
0.0724	1.0596	2.7658
0.2104	1.1976	2.6278
0.3484	1.3356	2.4898
0.4863	1.4735	2.3519
0.6243	1.6115	2.2139
0.7622	1.7494	2.0760
0.9002	1.8874	1.9380
1.0382	2.0254	1.8000
1.1761	2.1633	1.6621

7.4. DCT of 8×8 Pixels

Because of the corruption caused by the floating point arithmetic, instead of taking the DCT of the entire image, we can divide the image to 8×8 pixel squares and calculate the 2-D DCT of every square separately and write 8×8 output of the DCT in the corresponding places on the output matrix. When we want to calculate the IDCT, we do the same to the DCT matrix and we should get the original image. The following code does exactly that:

```
void DCTEight()
{
      ICBYTES RGB, grey;
      ICBYTES i, dcth, dctv;
      ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
      DisplayImage(FRM1, grey);
      CreateMatrix(dcth, grey.X(), grey.Y(), ICB_DOUBLE);
      CreateMatrix(dctv, grey.X(), grey.Y(), ICB_DOUBLE);
      DCT8x8(grey, dctv);
      IDCT8x8(dctv, grey, ICB_UCHAR);
      DisplayImage(FRM2, grey);
}
```

If you run this code, you will see exactly the same two grayscale mandrill images on the screen. You won't be able to notice any difference unlike when we applied the transform to the entire image in section 6.3.

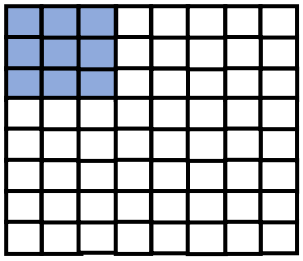
7.5. Image Compression

Let's add the following line to the DCTEight(.) function that we were introduced in the previous section:

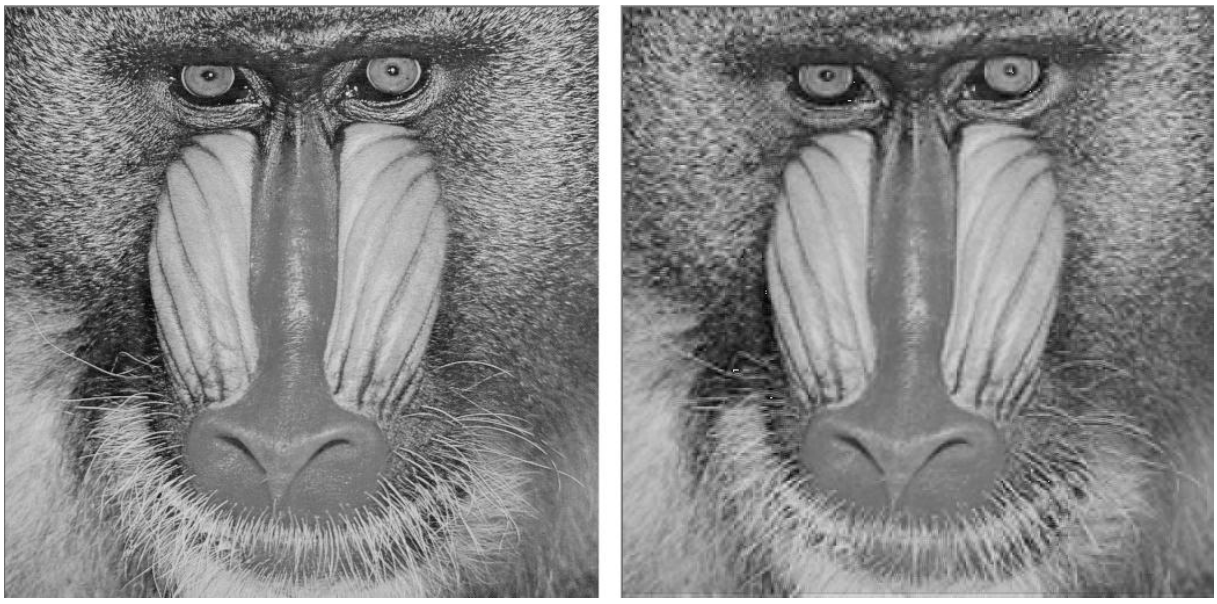
```
void DCTEight()
{
  ICBYTES RGB, grey;
  ICBYTES i, dcth, dctv;
  ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
  DisplayImage(FRM1, grey);
  CreateMatrix(dcth, grey.X(), grey.Y(), ICB_DOUBLE);
  CreateMatrix(dctv, grey.X(), grey.Y(), ICB_DOUBLE);
  DCT8x8(grey, dctv);
  SCANM(dctv) if ((x-1)%8 > 2 || (y-1)%8 > 2)dctv.D_[y][x] = 0;}}
  IDCT8x8(dctv, grey, ICB_UCHAR);
  DisplayImage(FRM2, grey);
}
```

```
for (int y = 1;y <= dctv.Y();y++) {
    for (int x = 1;x <= dctv.X();x++) {
```

What does this extra line of code do?



As shown with the purple row, that line of code expands to a pair of nested for loops that scan the entire DCT matrix. The modulus operations ((x-1)%8) and ((y-1)%8) ensures that in every 8×8 square in that matrix, the first 3×3 elements are kept (blue ones on the right) and the rest of the elements in that 8×8 square are set to zero. Note that this is performed before the DCT matrix is inverse DC transformed. In other words, we are deleting %86 of the information from the DCT matrix before reconstructing the original image back. Figure 7.4 shows that the loss of quality in the output image is barely noticeable. You have look really carefully to notice the differences.



Original

Compressed

Figure 7.4. The original grayscale Mandrill image and its compressed version after %86 of the DCT components are erased.

In Figure 7.4, we see that despite deleting %86 of the information from the DCT matrix, or in other words shrinking it down to %14 of what it was, there is barely any distortion in the inverse DCT image. As mentioned in section 7.2, that is because, after taking the DCT, most significant information stays at the beginning parts of the DCT signal. This is exactly how the JPEG (JPG files) accomplish image compression. Although it uses many other steps, reducing the number of DCT coefficients is the first step of JPEG compression.

8. The Affine Transform

If you take the picture of a rectangular table, what you will see is a parallelogram as shown in Figure 8.1. That is caused by your view angle towards the table and called perspective. If you have a security camera panning and tilting to find an object, you need to be prepared for the occurrence of that object from different perspectives in order to develop an algorithm that can recognize it. A rectangular table will almost never look like a rectangle in real life. It will look like a parallelogram.

Not only will it be a different geometric shape, its size will vary based on how close your camera is or how much zooming factor your camera uses. With a low zooming factor, or if your camera is far away, the table (parallelogram) will look smaller. On top of these, there is the rotation problem. If you start walking towards the other corner of the room, the parallelogram will appear rotated as shown in Figure 8.1. So how do we deal with so many variations? The answer is the Affine Transform.

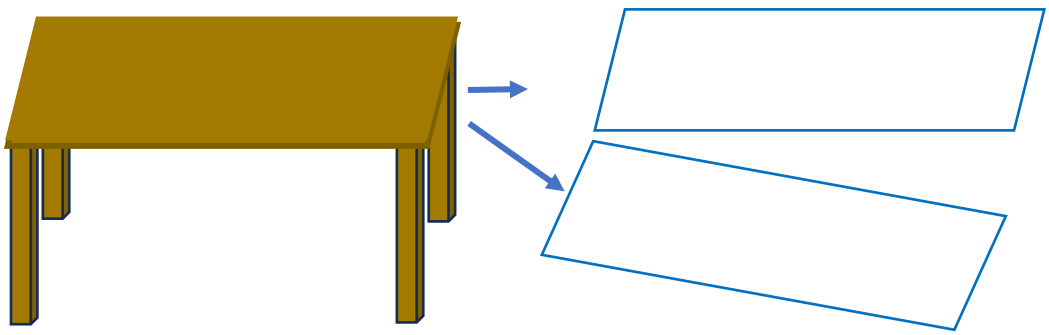


Figure 8.1. A rectangular table will look like a (rotated) parallelogram from your viewing angle.

8.1. Transforming Coordinates

Consider the picture of the table in Figure 8.1. We can define the surface of the table as a rectangle and obtain coordinates of each corner with respect to the top left of the image. Since the image will be stored in a matrix and the matrix coordinates start from the top left corner, this seems to be the logical choice. According to this coordinate system, the table will appear like a parallelogram. However, if we choose the reference point (the origin) as one of the corners of the room, and consider the tallness of the table the "z" coordinate, then the corner points of the table will appear as a rectangle.

This tells us that the perceptual changes are a matter of the coordinate system that we use and if we can find a way to translate from one coordinate to the other, we will solve our problem of recognizing the table from different angles. That translation is achieved by a simple matrix dot product:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

The x and y coordinates of the original coordinate system can be converted to the other coordinate system, (x_1,y_1), by simple cross multiplication of the original matrix with a 2×2 matrix as shown above. Now let's conduct some experiments using the I-See-Bytes library to see if this is true.

```
void Affine()
{
	ICBYTES img1, img2;
	ICBYTES rectangle = { {100,50},{250,50},{250,150},{100,150},{100,50} };
	CreateImage(img1, 400, 300, ICB_UINT);
	CreateImage(img2, 400, 300, ICB_UINT);
	for (int q = 1; q < 5; q++)
		Line(img1, rectangle.I(1, q), rectangle.I(2, q), rectangle.I(1, q + 1),
rectangle.I(2, q + 1), 0xff00);
	DisplayImage(FRM1, img1);
	ICBYTES affine = { {1.0,-0.5 },{0.0,1.0} };
	ICBYTES parallel, t, coor = { {0},{0} };
	CreateMatrix(parallel, 2, 5, ICB_INT);
	for (int q = 1; q < 6; q++)
	{
		coor.I(1, 1) = rectangle.I(1, q);
		coor.I(1, 2) = rectangle.I(2, q);
		t.dot(affine, coor);
		parallel.I(1, q) = t(1, 1);
		parallel.I(2, q) = t(1, 2);
	}
	for (int q = 1; q < 5; q++)
		Line(img2, parallel.I(1, q), parallel.I(2, q), parallel.I(1, q + 1),
parallel.I(2, q + 1), 0xff00);
	DisplayImage(FRM2, img2);
	DisplayMatrix(MLE, parallel);
}
```

If you run this program, you will get the drawings shown in Figure 8.2.

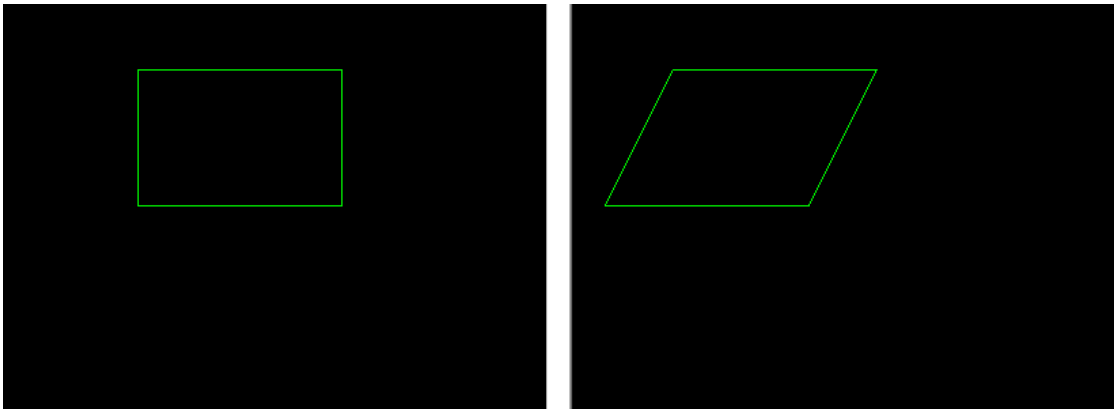


Figure 8.2. The output of the code that demonstrates the affine transform by transforming a rectangle into a parallelogram.

In this program, first we define the coordinates for 4 lines. A line requires two coordinates, one for start and one for end. Since the end coordinates of our lines are beginning coordinates of the next line, we need 5 coordinates to draw a rectangle in a loop:

```
ICBYTES rectangle = { {100,50},{250,50},{250,150},{100,150},{100,50} };
for (int q = 1; q < 5; q++)
            Line(img1, rectangle.I(1, q), rectangle.I(2, q), rectangle.I(1,
q + 1), rectangle.I(2, q + 1), 0xff00);
```

Next, we want to perform the following matrix operations to these coordinates:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x + y/2 \\ y \end{bmatrix} = \begin{bmatrix} 1.0 & -0.5 \\ 0.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

The affine transformation matrix shown above causes the objects to shear. In the previous code example, the affine transformation matrix was defined as:

```
ICBYTES affine = { {1.0,-0.5 },{0.0,1.0} };
```

The identity matrix as the affine transform matrix gives the same results:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Whereas, changing the corner values shrinks or stretches the image. If $\alpha>1$ it shrinks the image in the horizontal direction and if $0<\alpha<1$ it stretches in the same direction. Same goes for β in the vertical direction.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x + y/2 \\ y \end{bmatrix} = \begin{bmatrix} \alpha & 0.0 \\ 0.0 & \beta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

But probably the most useful outcome of the affine transformation is rotation. The following matrix causes the coordinates to rotate:

$$\begin{bmatrix} \cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

For example, cos(30) is 0.866 and sin(30) is 0.5. In the previous program if we set the affine matrix as,

```
ICBYTES affine = { {0.866,0.5 },{-0.5,0.866} };
```

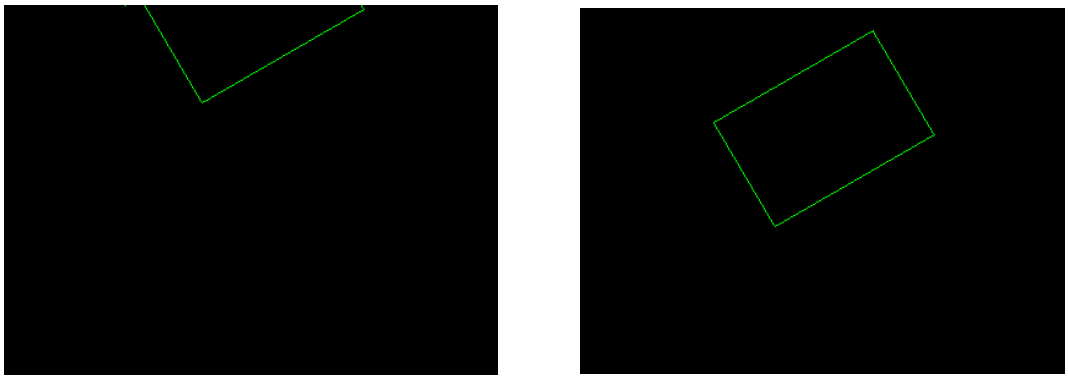
Then we will get the image shown in Figure 8.3-a. The rectangle has been rotated 30 degrees with respect to the origin, which is the upper left corner of the image. We can add to the y coordinates to bring the output image by adding 100 during the drawing just before the end of the function to obtain Figure 8.3-b.

```
      for (int q = 1; q < 5; q++)
            Line(img2, parallel.I(1, q), parallel.I(2, q)+100,
parallel.I(1, q + 1), parallel.I(2, q + 1)+100, 0xff00);
      DisplayImage(FRM2, img2);
      DisplayMatrix(MLE, parallel);
}
```

In order to include such coordinate shifts, which is also known as translation, the affine transformation can be express as follows:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + x_t \\ cx + dy + y_t \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & x_t \\ c & d & y_t \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

However, the I-See-Bytes library expects the affine matrix to be a 2×2 matrix as input argument to functions because translation is a trivial task and can be easily accomplished manually by the user.



(a) (b)

Figure 8.3. (a) The rectangle in Figure 5.2 after 30-degree rotation with respect to the top left corner. **(b)** Same rectangle shifted 100 pixels downwards.

The affine transform can be used to create symmetric versions (mirroring) of coordinates through any axis.

8.2. Affine Transform on Images

The following code demonstrates how to perform affine transform to the pixels of an image:

```
void AffineImage()
{
  ICBYTES RGB, inp;
  ReadImage("playbot.jpg", RGB); Luma(RGB, inp);
  ICBYTES outp, region = {240,140,170,100};
  double theta = -25.0;
  ICBYTES affine = { {cos(3.1415* theta /180.0),sin(3.1415 * theta /
180.0)},{-sin(3.1415 * theta / 180.0),cos(3.1415 * theta / 180.0)}};
  if(!AffineNearest(inp, affine, outp,region))
       ICG_printf(MLE,"Couldn't!\n");
  DisplayImage(FRM1, inp); DisplayImage(FRM2, outp);
}
```

When we closely look at this code we see that a vector called region is defined:

```
ICBYTES outp, region = {240,140,170,100};
```

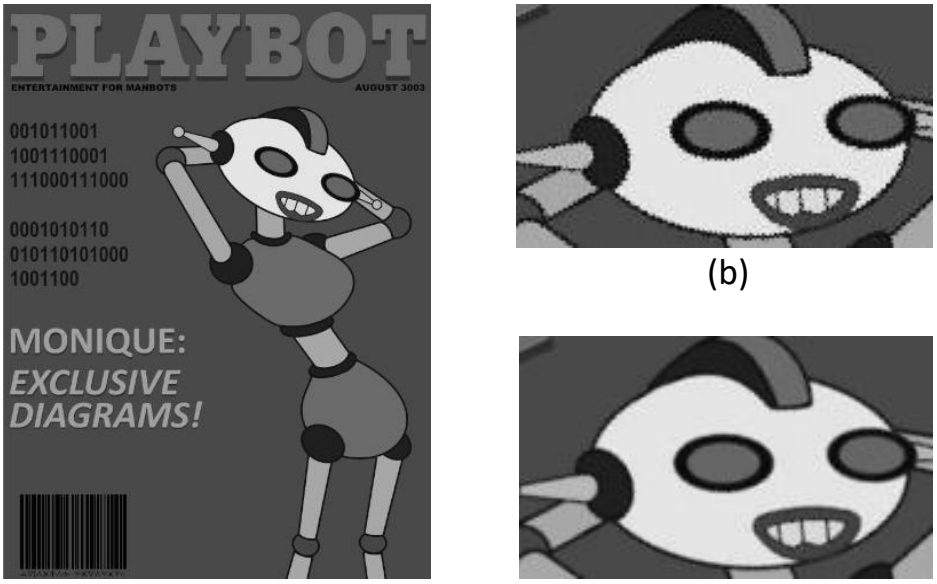
This vector's first two elements hold the center point where the image will be rotated around. Next two elements (170 & 100) are the width and the height of the output image rectangle, or the image patch, we wish to cut out of the big picture. Next, we see that the angle of rotation, theta, has been defined as -25 degrees. Then the affine matrix is created:

```
ICBYTES affine = { {cos(3.1415* theta /180.0),sin(3.1415 * theta /
180.0)},{-sin(3.1415 * theta / 180.0),cos(3.1415 * theta / 180.0)}};
```

Which is essentially the following matrix: $\begin{bmatrix} \cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$ where θ is -25 degrees converted to radians because the sin(.) and cos(.) functions require that their input arguments must be in radians. Once the input image and the parameter s are ready we call the following function:

```
AffineNearest(inp, affine, outp,region);
```

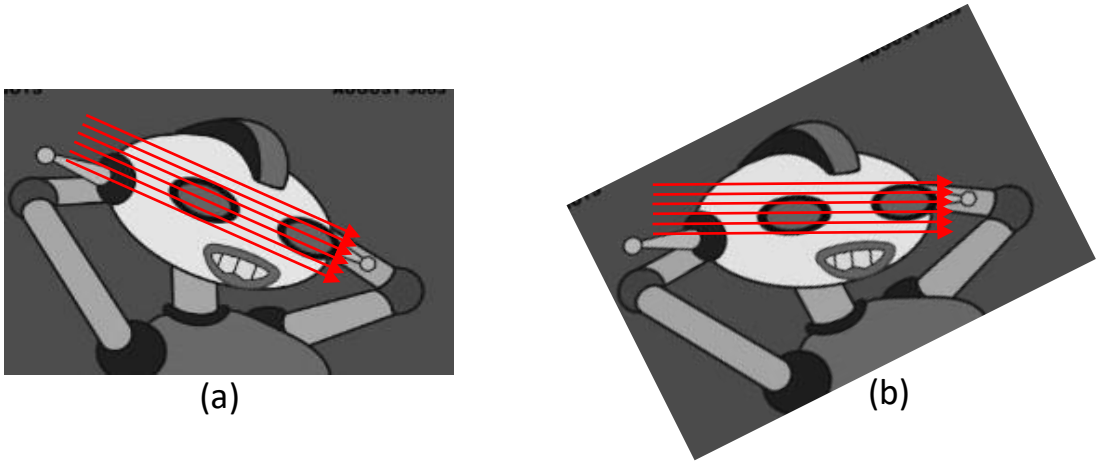
Where "inp" is the target image and the "outp" is the rotated region of the target image targeted by the "region" vector and rotated by the "affine" matrix. The result is shown in Figure 8.4.



(a) (b) (c)

Figure 8.4. (a) The face of the robot Monique as the target image. **(b)** The face of Monique (magnified) tilted -25 degrees using the nearest neighbor interpolation. **(c)** The face of Monique (magnified) tilted -25 degrees using the bilinear interpolation.

So how do we rotate an image? What we do is we take the original image and start sampling it at a different angle as shown below. You may think of this as if we print out the image and then rotate the picture 25 degrees and then using a scanner scan the rotated picture so that the head looks levelled as shown in Figure 8.5-b. Unfortunately, the image signal is a discrete 2-D signal. If it were a continuous signal, we could have sampled it in any direction we want. But with



(a) (b)

Figure 8.5. (a) In order to rotate an image, we need to sample that image at a skewed angle. **(b)** That is almost like printing the image and then rotating and scanning the picture using a scanner.

a discrete signal, our sampling angle causes our samples to fall in between the pixels.

Figure 8.6 shows the locations of the new (red) grid for tilted scanning. The positions of the new pixels (red dots) are calculated using the affine transform. How do we determine the level of those pixels? In order to do that, the simplest thing we can do is select the nearest neighbor. An example is shown on the right side of Figure 8.6. The new pixel location is closest to the pixel C. Therefore, we can choose the pixel value of C for our new (x,y) location. However, doing so makes the resultant picture look rugged as shown in Figure 8.5-b. If your image is high resolution, that might not be noticeable by the human eye. However, if you wish to apply image processing algorithms such as edge detection afterwards, it will create problems.

A better approximation can be achieved by using the bilinear transform. This transform assumes that our new sampling point is on a square surface whose corner eleveations are defined by the four pixel points, A, B, C and D. In order to calculate the pixel value of the red dot at (x,y), we first calculate p1 and p2. That is very easy:

$p1=Bm+A(1-m)$ and $p2=Dm+C(1-m)$

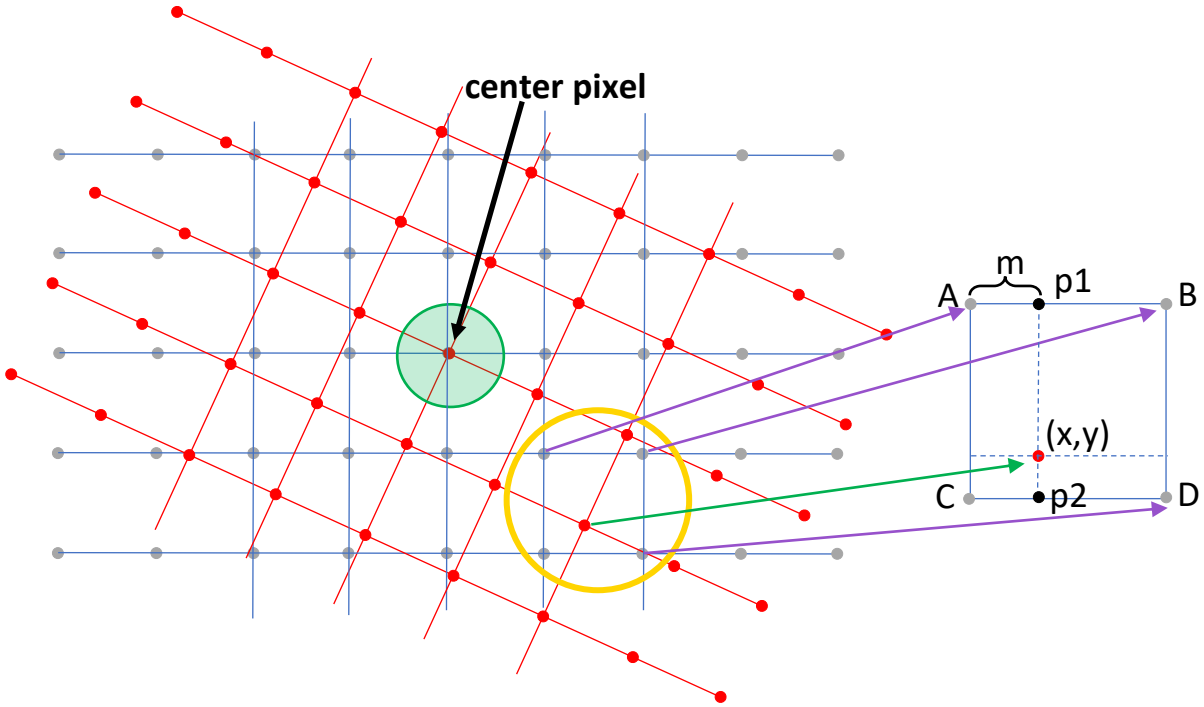


Figure 8.6. The red dots are the location of the new sampling positions. For the nearest neighbor interpolation, the pixel level (C) is selected because it is closest to the new sampling point (x,y).

Then we calculate the weighted average of p1 and p2 using the distance of y of our point from these points. This approximation is much more lengthy and processing power consuming then the nearest neighborhood method but it produces smoother results. In the I-See-Bytes library, all we have to do is call the

```
AffineBilenear(inp, affine, outp, region)
```

function, the same way we called the AffineNearest(.).

9.Machine Learning

Imagine that we wish to build a device to identify the gender of university students. Let's assume that we don't know computer vision techniques. How can we do that? We can put sensors at the entrances of classrooms that measure the weight of a student and then decide what the student's gender is. But in order to make those decisions correctly we need to know statistical information about the weights of male and female students. Therefore, our next step will be to measure and record weights of say 50 students from each gender and create a weight distribution histogram for each gender as shown in Figure 9.1.

Unfortunately, these weight distributions hardly give us any useful information. For example, if the student weighs between 55-65 kg, the chances of student's being girl is %55 and a boy is %45 because there are almost same number of students in our survey group among boys and girls in that category. On the other hand, if a student weighs between 35-45, then we can predict with more confidence that it's a girl because in that category the number of girls is 6 times that of boys. Or if the student weighs more than 85 kg, it is most probably a boy.

In Machine Learning (ML) terminology, boys and girls are called **classes** and the measure we use to identify them, in our case their weights, is called a **feature**. Each of the student (or their features) is called a **sample**. The survey

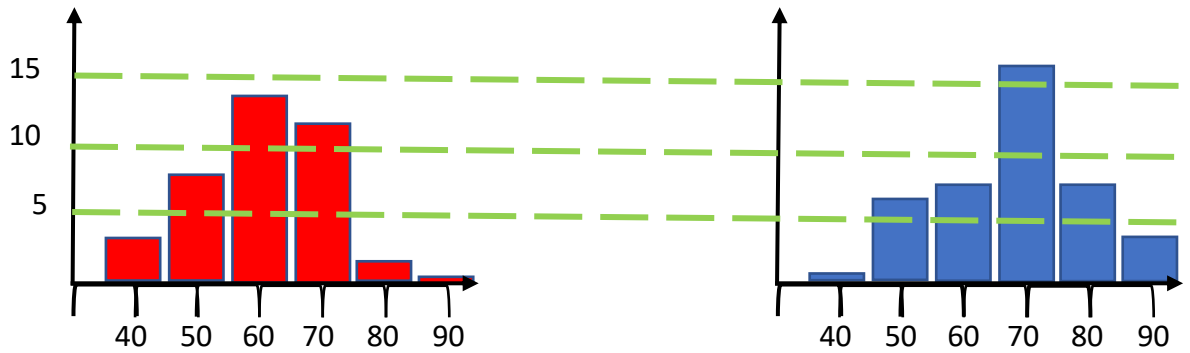


Figure 9.1. Weight distribution of 50 female (red) and 50 male (blue) university students.

group of 50 students from each gender is called the "**training set**." The algorithm that decides if a student is a boy or a girl by analyzing or looking at the training set is called a **classifier**. The mechanism that creates the weights of these (or weight distribution) students is called a **process**. In our case, the process would refer to the effects of (the ratio of) testosterone (male hormone) and estrogen (female hormone) on human body.

9.1. Gaussian Distribution

When we look at the shapes of the histograms in Figure 9.1, we notice that the histograms are shaped like a triangle. That is because each class has majority of samples that accumulates around a value. For example, for girls, majority is between 55 and 65 kilograms while for boys it is between 65 and 75 kg. That is because the (biological) process that creates the weight of the students has a hidden target value. However, because this process is not perfect, it sometimes overshoots and sometimes undershoots creating a weight distribution around this target value. In ML terminology, this target value is called **mean** or "**expected value**."

Scientist have discovered that nearly every natural process creates a unique distribution. Obviously, knowing the distribution of our data can help us classify unknown samples. The most common distribution that is encountered for natural (daily life) processes have a distribution called Gaussian. Imagine that you are throwing darts at a target as shown in Figure 9.2. Obviously, you can't always hit the bullseye. If you are good at this, then most of the darts that you throw will be around the bullseye but there will be few others away from it. If we measure the distance between every dart stuck on the target board and the

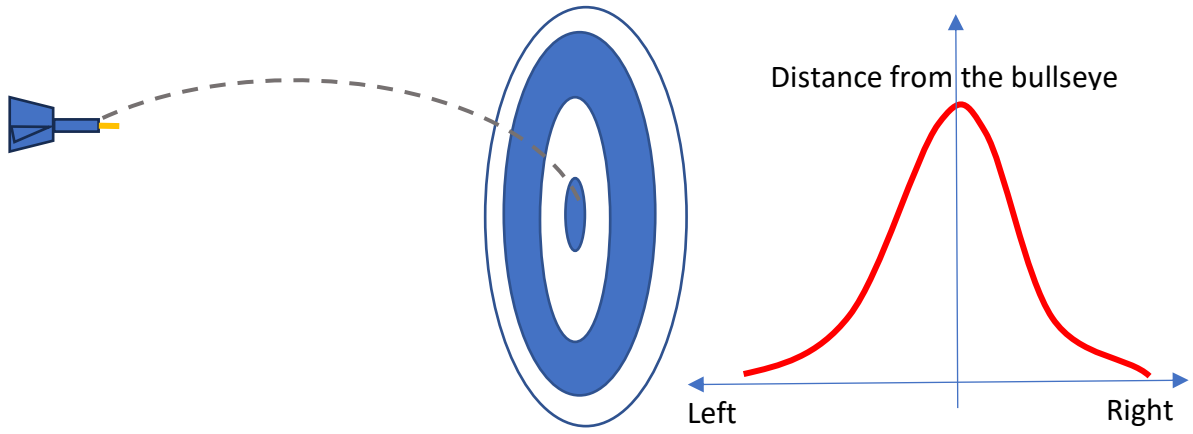


Figure 9.2. When a person throw darts at a target, the distance distribution of the darts from the bullseye creates a shape as shown with the red curve.

center of the target, and create a histogram including whether the dart hit the left (or bottom) or the right (top) of the bullseye then we would get a distribution as shown with the red curve. That curve is the shape of the Gaussian distribution:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

In this function, μ is the mean (or the expected value or **expectation**), and sigma (σ) is the **standard deviation**. Note that the square of standard deviation is called the **variance**. The mean decides where the peak point of the curve shall be and the variance decides how flat the curve is. The Gaussian distribution is also called the **normal** distribution. The other most used distribution is the uniform distribution where every outcome has the same probability. The shape of that distribution looks like a rectangle.

9.2. Academic vs. Industrial Approach

Having worked both in the industry and the academia for decades, the author of this book witnessed fundamental differences between the academia and the industry when it comes to ML. In the academia, machine learning research has one major goal: To increase the correct classification rate using sophisticated classifiers. As mentioned in the first chapter, academicians are not concerned with the efficiency or the price of the implementations of their algorithms. They mostly care about how successful their algorithms are because it is very unlikely that they can publish articles narrating how little effort they spent to solve a problem. They need to prove that their algorithms and methods can classify genders of more students correctly compared to that other guy. In order to achieve that, they use as many different features as they can. For example, in the previous sections, we used only the weight of the students. They on the other hand, will use their height, shoulder and waist width, hair color and so on to classify students. That way they can increase their chances of correctly classifying the gender of the students. This approach is not suitable for the industry because collecting all that information, building devices that measure that information at the entrance of the class room and writing the software would cost a lot.

They also focus on using the fanciest, newest classifier algorithms because the more sophisticated the algorithm they use the more likely their papers will

be accepted for publication. There are thousands of ridiculously complicated articles in the literature for performing simple, mediocre tasks. (A list will be published at the end of this book in the future.) As we discussed in the first chapter, such approaches would kill the competitiveness of a company causing it to go bankrupt. Therefore, good engineers must avoid such approaches.

Figure 9.3 demonstrates how these two approaches differ. Because the academicians chose to include so many standard (commonly used by other researchers) features, naturally, some of those features are inseparable as shown in Figure 9.3-a. When the distributions of the features of two classes are so close that they are almost on top of each other, they are called inseparable. Most of the time the dataset of the problem they work on is published on the internet and the results are discussed using this data set, therefore, it is possible improve the results %2-3 compared to some other researcher's paper. They alter the parameters of classifier algorithms and apply methods such as boosting, ensemble learning etc. to come up with a slightly higher success rate (accuracy). The fact to the matter is, when the size of the data set is finite and they are hardly separable, these approaches might be improving the results by **pure luck**. The same approach might do %5 worse on a completely new data set. In real life the data set is infinitely large therefore such approaches are **not dependable** for the industry.

What the industry does is research separable features (Figure 9.3-b) instead of researching sophisticated classification and training techniques. While the academicians research mathematical ways of separating standard (commonly used) features, the industry must work like a detective to find clues that separate these classes. However, when the industry finds such features, they never publish their results because first of all such information is commercially very

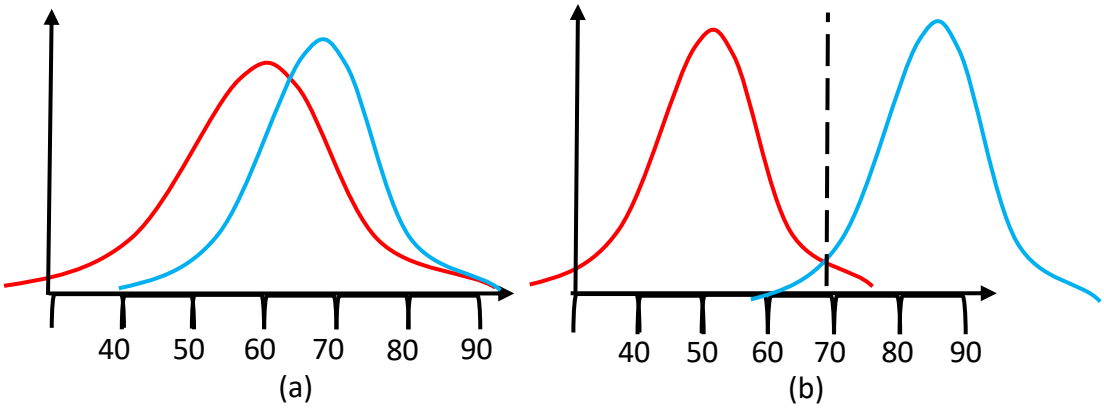


Figure 9.3. The distribution of the features of different classes. (a) Inseparable case. (b) Separable case.

valuable and you wouldn't want other companies to find out your engineering secrets.

Secondly a separable feature makes the problem mathematically too simple. **Too simple to be published in a scientific journal**. What the editors and the referees want to see is how bravely you fought against the evil forces of inseparable features and won. If you invented a completely new feature that classifies %99 samples correctly using the dumbest classifier, where is the fun at reading that? Consider the boy and girl students problem. You can use their weight, height, limb length, waist and butt size, shoulder width. These are the common features one would think of. With those features you reach %95 success rate at most. On the other hand, if a company invents a simple detector that measures the testosterone/estrogen ratio of a student, the game would be over. We shall call such a feature "**the golden feature**" because one simple feature is capable of correctly classifying more than %99 of the samples correctly.

A little-known fact about the human foot is that women's feet are wider compared to their length than men's. Creating a sensor that measures the width and the length of students' feet when they step on it would be very easy. Once you receive the length and the width of the student's feet, all your algorithm has to do is calculate the ratio and apply a threshold. A **THRESHOLD**? If you have only one feature, the only classifier you need is a threshold. Or maybe two as shown in Figure 9.4-b. Rarely, you may need more but you get the picture. The important thing is, the industry's products are a lot more reliable than that of the academicians. In countries with well-established culture of industrial research, the industry never outsources its research **directly** to academicians.

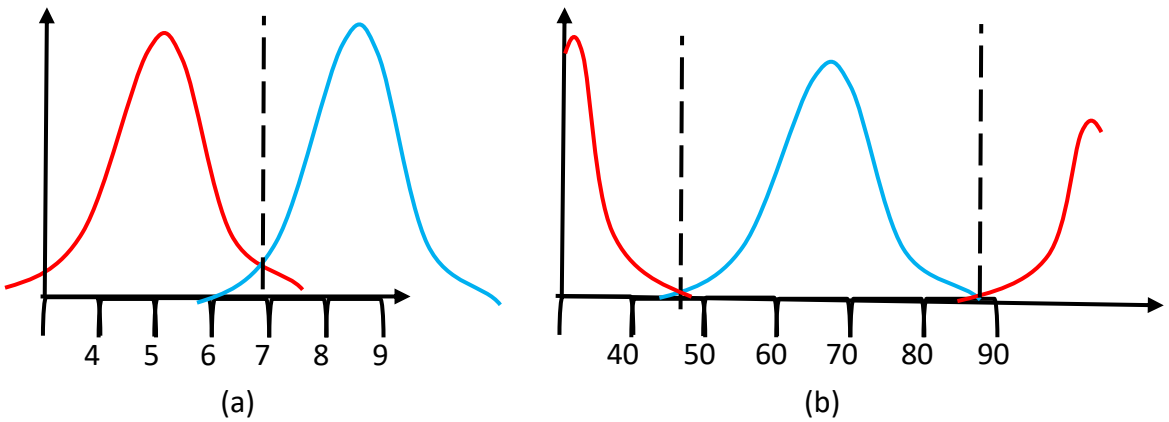


Figure 9.4. If you have a separable feature as shown in (a), all you need is a single threshold or two thresholds if as in (b).

9.3. The Case of Multi Features

So, what happens when we use more than one feature? What if we also measure the height of the students along with their weight? If we mark each student's weight and height on a chart, then Figure 9.5-a is what we would get. In this two-dimensional plot, we see that the marks belonging to the boys (blue ones) are towards the upper right and the ones belonging to the girls (red ones) are toward the bottom left.

In this example there are 10 boys and 10 girls to simplify the plot. If we apply a threshold to the weights (represented by the dashed green line), then we can correctly classify 8 of the girls and 9 of the boys making our success rate (accuracy) %85 as shown in 9.5-b. If we apply a threshold to the heights (represented by the dashed purple line) then we could correctly classify 7 of the girls and 7 of the boys making our accuracy %70. What if we could put a threshold on both of the axis instead of one at a time? That would require the equation of a line:

$$y=ax+b$$

If we use the line represented by the dashed orange line, then we could correctly classify 9 of the girls and 9 of the boys, giving us an accuracy of %90 which is higher than putting a threshold on either of the axis. This proves that we can increase the success rate of our classification by increasing the number of our features.

If we add one more feature, for example the shoe size of the students, then perhaps we can further increase the success rate of our classifier. But with three features, we would need a 3-dimensional space to plot our data. That gives us

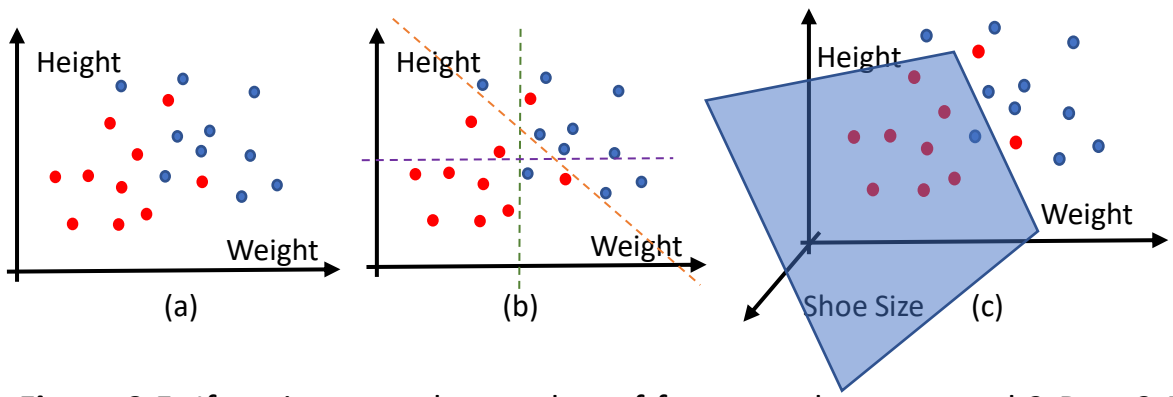


Figure 9.5. If we increase the number of features, then we need 2-D or 3-D spaces to represent our data allowing more options for thresholds.

more freedom to put thresholds. Instead of lines, now we can use surfaces as threshold as shown in 9.5-c.

9.4. Types of Classifiers

So far, we have used thresholds to separate different classes (boys & girls). Each threshold can be considered a boundary for that class. But a boundary need not be a line. It can have curves. Consider the boundary shown in Figure 9.6-a. If we had a classifier that has the boundary shown with the green dashed line, then we could classify %100 of the samples correctly. There are classifiers that can calculate such borders efficiently and one of them is helping you learn already. The (Artificial) Neural Networks (ANN) work exactly that way. Note that, although the examples shown in Figure 9.6 are 2-dimensional, the reader must envision them happening in higher dimensions as well. Another popular classifier algorithm that calculates such borders is the Support Vector Machine (SVM).

The second type of classification method is to use a probabilistic field. You can imagine a field that is emanating from a center (or central area) and slowly diminishing nonuniformly as you go further from that center as shown in Figure 9.6-b. You can imagine as if boys have a blue probabilistic field and girls have a red probabilistic field covering the entire feature space. In order to decide whether a set of features belong to a girl or a boy, we calculate the intensity of the probability field for both genders and then assign it to the one that has higher intensity. Examples of such classifiers are Bayesian and Parzen Window classifiers.

Finally, the third type of classifiers work based on the distance. The mean, or the expectation of the features for each class is marked on the feature space as shown in Figure 9.6-c. The closer a sample is to this point, the more likely it is that, this sample belongs to that class. Examples of classifiers using this approach

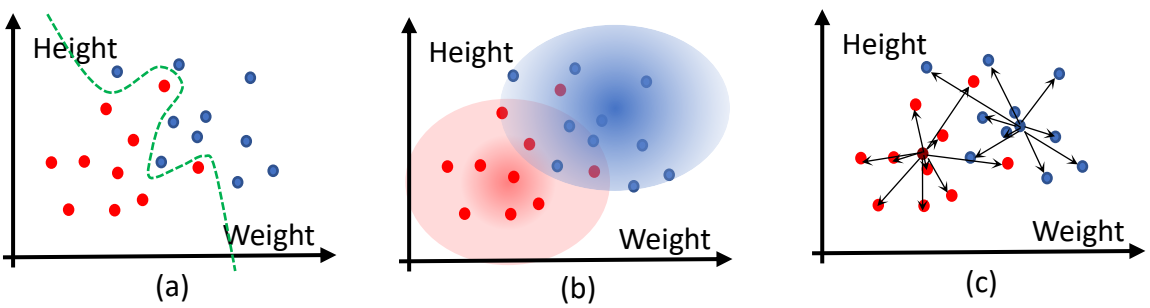


Figure 9.6. A Boundary type (a), a probabilistic field type (b) and a distance type (c) classifier.

are the Minimum Distance Classifier, Mahalonobis Distance Classifier and K-means.

Among these approaches, the probabilistic and the distance classifiers have one advantage over the boundary classifiers: They also give you a (linear) measure of how strongly a sample belongs to a class. In other words, they give you an estimate of their confidence. Therefore, if you want to be very precise, you can choose not to classify samples (undecided state) that are not a strong member of a class (process). Such a property can be very important for classification problems where misclassification has fatal consequences. For example, if you want to classify currency denomination, that is if you want to figure out if a banknote is 5, 10, 20, 50, 100 etc., then you need these types of classifiers because if you classify 1 dollar as 100 dollars then your currency counting machine will make the bank lose 99 dollars every time it misclassifies a banknote. Probabilistic classifiers are also more successful at classifying natural processes especially if the training set is small.

On the other hand, the success rates of the boundary classifiers are higher because they can model more complicated distributions. To sum up, the probabilistic and the distance classifiers can be made more precise because precision is defined as true (T) or correct estimations, divided by all (true and false (F)) estimations.

$$Precision = \frac{T}{T + F}$$

You can increase your precision by tightening the membership constraints. The boundary classifiers are more accurate meaning that they can classify more samples correctly but their wrong estimations can be higher:

$$Accuracy = \frac{T}{All\ Samples} = \frac{T}{T + F + U}$$

In this definition “U” stands for “Undecided.” Boundary classifiers such as the ANNs can give you a measure of how confident they are about their classification. Unfortunately, because these classifiers use nonlinear functions, their estimations are not always reliable. They can be made more precise by changing the training parameters. In the industry, “undecided” can be rephrased as “no call” and false positive can be called “miss call” because not everyone is familiar with concepts such as “accuracy” and “precision.” You may have to dumb down your sentences so that the “managers” can understand you. 😊

9.5. Features, Samples, Classes and Labels

In order to start using classifiers, we need to learn some of the common terminologies used in machine learning and how to implement / represent them in ICBYTES. As we mentioned before, a feature is one of the measurable outcomes of a process. The testosterone and estrogen hormones cause a human being to grow as a male or a female. Therefore, the underlying process is the hormones. The measurable features are weight, height, shoe size etc. If we write all of the features belonging to a student as a vector:

S = [78.3kg, 173cm, 40] or simply, S = [78.3, 173, 40] is our feature vector. If this student is female, we label this vector as [F], and if male then [M]. Features of the entire class can be represented as:

$$\underline{Samples} \qquad\qquad \underline{Labels}$$
$$\begin{bmatrix} 78.3 & 173 & 40 \\ 61.2 & 166 & 35 \\ 83.1 & 182 & 42 \end{bmatrix} \qquad \begin{bmatrix} M \\ F \\ M \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

If we write the feature vectors of each student one after the other, we get the “samples” matrix. We have to label these features so that we know which vector belongs to a girl or a boy. Therefore, we need another vector that contains the labels. If we write these on paper, we can use “M” or “F” but inside the computer everything is a number and the ICBYTES library requires them to be labeled as positive integers. In that case all the boys are in class-1 and all the girls are in class-2. If we have a feature vector and we don’t know which class it belongs to, we label it as zero (0). The goal of machine learning is to classify unlabeled vectors by studying the labeled ones.

The ICBYTES library needs you to prepare your data as shown above; one double type matrix for features and one int type vector for labels or class membership.

9.6. Mean and Covariance

We all know what mean is. It is the average of one feature inside the “Samples” matrix. For example, if we want to calculate the average height of the boys, then we need to sum height of all the features belonging only to boys and then divide it by the number of boys in those samples. But what is covariance? Covariance is the dependence of one feature to the other. For example, if a person is tall, usually their shoe size is larger. Covariance tells us which features

are independent. Independent features are more valuable for classification because dependent features carry less information. If we can predict the shoe size of a person by his/her height then why would we spend any effort to measure the shoe size?

In order to better understand what covariance is, we shall use the "Classifier" project that comes with the ICBYTES library. Figure 9.7 below shows a screenshot of this application. This program creates pseudo random feature vectors and then uses different types of machine learning algorithms to classify them. The top sliders are used for setting the mean of the random samples and the covariance matrix is used to define the covariance relationship between the features. In this example, the first element of the matrix is set to 5.0 meaning that the first feature, in our case "x," can fluctuate (approximately) ±5 around its mean which is 4.00. Note that both the x and the y variables have Gaussian probability distribution. The last value in the covariance matrix, 1.0 means that the "y" feature has a covariance of 1.0 therefore its fluctuation is less. Two 3.0 values at the corners of the matrix means that these two variables, x and y are dependent on each other strongly, and if one of them increases, the other one tends to increase and vice versa. If you set the sliders and the covariance matrix as shown below and press the "CREATE TEST SAMPLES" button, you should get a similar plot.

Because the x and y variables tend to increase together (not always but most of the time), the spread of the samples is diagonal on the plot.

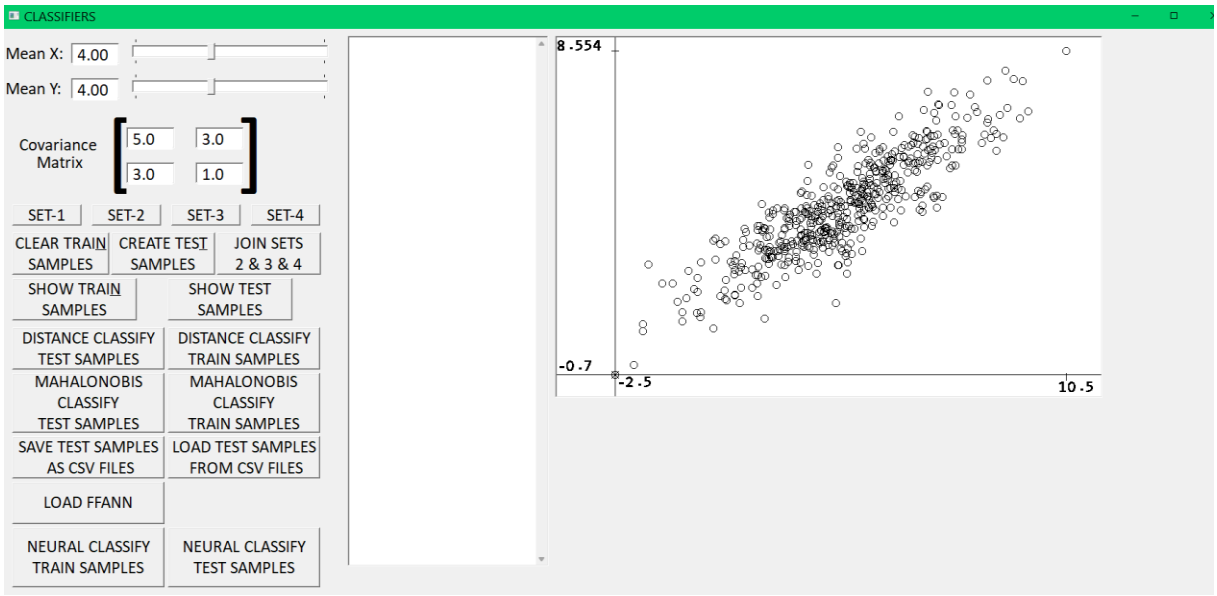


Figure 9.7. A screenshot of the "Classifiers" project that comes with the ICBYTES RV library demonstrating the effects of the covariance matrix.

In this project, the test samples are held in the "testset" matrix. You can find out what the covariance and mean of these samples are by calling the following function:

```
ICBYTES outcov,mean;
bool Covariance(testsamples, outcov, mean);
```

sending the "mean" vector is optional. If you only want the covariance calculated, then you can call this function as shown below:

```
Covariance(testsamples, outcov);
```

9.7. The Distance Classifier

As the name implies, the distance classifier categorizes samples based on their distance to some point and that point is the estimated mean of the classes. We first calculate the mean (center) of the training set, and then based on that mean, we calculate the distance of every sample:

$$Dxy = \sqrt{(x_m - x)^2 + (y_m - y)^2}$$

If you have only one set, then you can say "if $D_{xy} < D_T$ that sample is in." In other words, you put a distance threshold. If you have more than one class, then you need to calculate distances for each center (mean) and chose the center that the feature vector is closest to. Let's perform an experiment using the "Classifier" project. Using the GUI, create two sets one with means and covariances as shown below:

$$m_1 = (1.1, 1.1) \quad C_1 = \begin{bmatrix} 3.0 & -3.0 \\ -3.0 & 3.0 \end{bmatrix}$$

$$m_2 = (7.0, 7.0) \quad C_2 = \begin{bmatrix} 3.0 & 3.0 \\ 3.0 & 3.0 \end{bmatrix}$$

In order to do that, you need to adjust the sliders until they reach the mean values and you type in the values in the covariance matrices. Then first you press the "SET-1" button, then after adjusting the values again you press the "SET-2" button. If you did this correctly, you should get a plot similar to what is seen in Figure 9.8. Remember that the random number generator creates different numbers every time so your plot will be similar but not exactly the same.

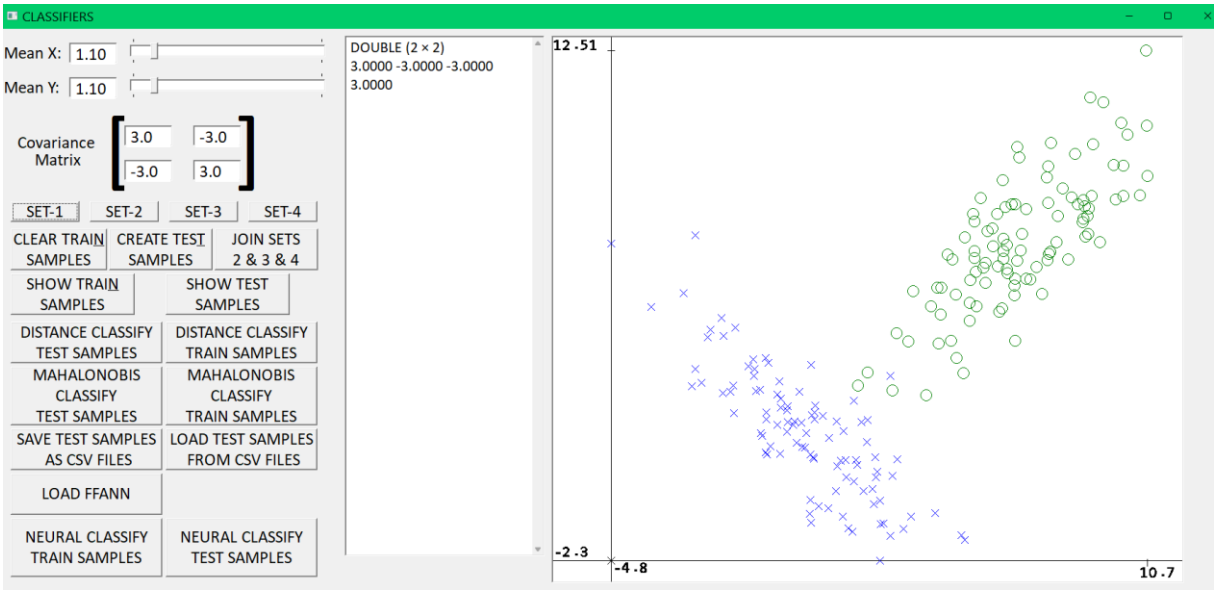


Figure 9.8. Two sets of samples generated by the program. Features of the first set is marked with blue crosses and the second one is marked with green circles

After creating these two sets, if you press the “DISTANCE CLASSIFY TRAIN SAMPLES” button, you will get the plot shown in Figure 9.9. You can clearly see that the distance classifier misclassified some of the samples which are encircled with a red ellipse. Those samples inside the red ellipse should have been marked with green color because they belong to the second set. Unfortunately, because they are closer to the center of the first set, 6 of them have been (mis)classified as members of it.

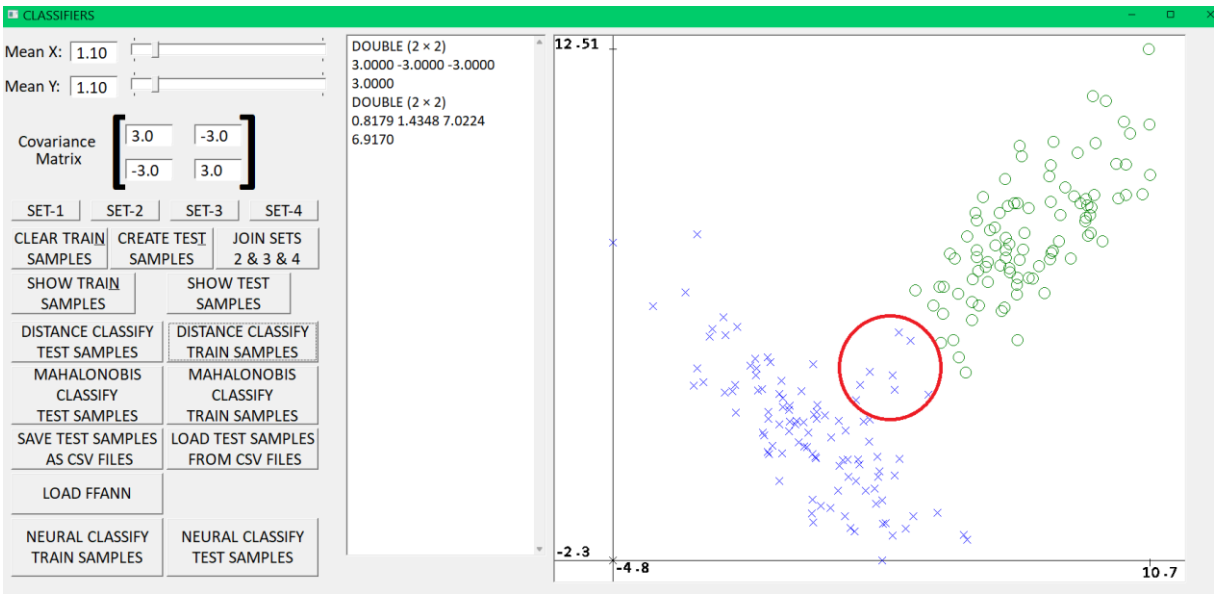


Figure 9.9. The result of the distance classifier. The samples shown inside the red ellipse are misclassified.

So, how does the ICBYTES library perform distance classification? Here is the function that runs when you press the “DISTANCE CLASSIFY TRAIN SAMPLES” button:

```
void DistClassifyTrain()
{
      ICBYTES means, labels;
      FindSetMeans(samples, members, means);
      DisplayMatrix(MLE, means);
      DistanceClassifier(samples, means, labels);
      Plot(img, samples, labels, 50.0, 0xffffff, 10);
      DisplayImage(FRM1, img);
}
```

The first thing we need to do is calculate the means of each feature in each set. We use the FindSetMembers(.) function to do that. The “samples” matrix contains the features of both set-1 and set-2 and the “members” vector contains their labels. This function shall return you the “means” matrix which contains the following values:

$$means = \begin{bmatrix} 1.1332 & 1.0510 \\ 6.8149 & 6.6961 \end{bmatrix}$$

This means that the mean values of first set’s features are (1.1332, 1.0510). Remember that when creating the set-1, we had set the sliders to (1.1, 1.1). As a result, the randomly generated samples (100 of them), have a mean very close to that value. According to the matrix above, the mean of the set-2 is (6.82, 6.7) which is very close to our initial mean setting of (7.0, 7.0).

After calculating means of each set, all we have to do is call the DistanceClassifier(.) function and let it create a new “labels” vector as the result of the classification. Rest of the commands plot these samples according to the new labels generated by the distance classifier. The plot function expects at least four arguments as input. The first one, ‘img’ is the image matrix that the graph will be drawn. The second matrix contains (x,y) coordinates, and the third one, ‘labels’ contains which group those (x,y) coordinates belong to. Finally, the zoom factor is required. Optionally, the background color and the marker size can be set as well.

9.8. The Mahalonobis Distance Classifier

In the previous section, we witnessed the shortcomings of the distance classifier. It misclassified 8 samples because it measured the straight distance from the centers of the sample sets without taking their covariance into account.

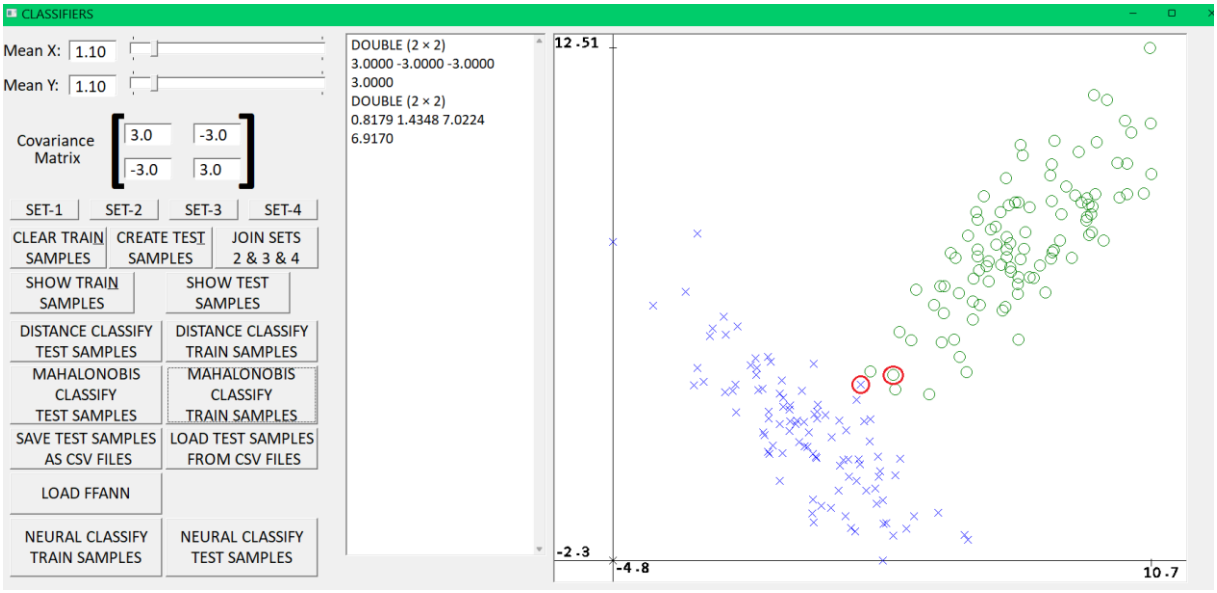


Figure 9.10. The Mahalonobis distance classifier correctly classified 4 of the 6 samples that were misclassified by the distance classifier.

The Mahalonobis distance classifier was invented exactly for that purpose. The Mahalonobis distance classifier is defined as:

$$D_M = \sqrt{(F - m_S)^T C^{-1} (F - m_S)}$$

Where C^{-1} is the inverse of the covariance matrix and m is the mean of the set.

Figure 9.10 shows the result of the Mahalonobis distance classifier. If you press the "MAHALONOBIS CLASSIFY TRAIN SAMPLES" button you should get a plot similar to that. Among those 6 samples encircled by the red circle, 4 more of them were classified correctly compared to the distance classifier. The other 2 are irrecoverable because they are very close to the center of the opposing sets. This example shows us two sets that are almost completely separable but not exactly. And even though the Mahalonobis distance classifier is a simple algorithm, for this case even the most sophisticated classifiers such as Deep Neural Networks or Support Vector Machines would fail on these samples as long as the training and the test sets are different. (Created by the same process but different sets.) Proving that there is a limit to the success rate of classification if the data is nonseparable.

So how do we perform Mahalonobis distance classification using the ICBYTES library? The MahalClassifyTrain(.) function shows how it is done:

```
ICBYTES set, mean[4], covariance[4], dist[4];
for (int i = 0; i < 4; i++)
```

```
{
      ExtractSetFromSamples(samples, members, i + 1, set);
      Covariance(set, covariance[i], mean[i]);
      Mahalonobis(samples, mean[i], covariance[i], dist[i]);
}
```

Because the GUI creates up to 4 different classes, we need to separate features belonging to each set before calculating their covariance and mean. The ExtractSetFromSamples(.) function does that by looking at the membership vector. The features belonging to labels (sets) 1, 2, 3, and 4 are extracted into the "set" matrix one by one and then send to the Covariance(.) function. This function calculates the mean and the covariance matrix for each of the sets and places in the array of fluids mean[4] and covariance[4]. Then we call the Mahalonobis(.) function to calculate the square of the Mahalonobis distance. Note that this function calculates the **square** of the distance and stores them in one of the dist[4] vectors, not the actual distance. Why doesn't it find the square root? The answer is to be fast. Since we are going to compare these four (in our case two because we have only two sets) distances and select the shortest (smallest) one, there is no need to calculate the square root and spend our precious processing power needlessly. The ICBYTES library is designed for both speed and comfort. If A>B then the square root of A is also larger than the square root of B. If you need to set a threshold to further eliminate the outliers, then you should use the square of the threshold value you had in mind.

The rest of the MahalClassifyTrain(.) function code processes these dist[4] vectors and decides which set the sample is closest to and then labels it accordingly and plots it on the screen.

9.9. The Feed Forward Artificial Neural Network

There are many types of Artificial Neural Networks (ANN) and the Feed Forward (FFANN) ones are the most common type used for classification. They are so common that most of the time when we say that we use ANNs, that means that we actually use FFANNs. The topology of these networks is shown in Figure 9.11. There is always an input and an output layer and there may be several hiden layers. Each round shape is called a node and all of the arrows represent signals (values) that are being transferred to the next layer nodes after being multiplied by weights. All of the hidden layers and the output layer have their associated weights for each of the arrow that is pointing to them. These nodes also have biasses which are not shown in the figure. The biasses are negative or

positive numbers that are added to the value of that node. Inputs are multiplied with the weights of each node and summed, with the bias as well, and then the presented to the next layers through their weights. Input values travels from left to right towards the output. That is why this type of neural network is called Feed Forward.

At the end of each node, there may be a limitter, a hard limiter or a soft one such as the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

On top of all of these there may be a normalization function at the inputs and outputs adjusting hem in the range of (0,1) or (-1,1).

The ICBYTES library holds the entire parameters of an FFANN inside a vector list. However, at the moment the ICBYTES library does not allow training of the ANN. That is because the ANN field is a rapidly developing field with new topologies and different “limiter” or “transfer” functions continuously being invented causing the training algorithms to be reinvented as well. It is not possible for us to keep up with these rapid advancements. Therefore, instead of including numerous training functions for countless ANN topologies, the ICBYTES library allows the user to transfer the training data to specialized ANN software and then load the neural network back to the ICBYTES library.

9.10. Exporting Data to the MATLAB

The ICBYTES library has various methods for exporting data. The most suitable method for exporting the training data to Matlab software is to use the CSV (Comma Separated Values) files. We will demonstrate how this is done

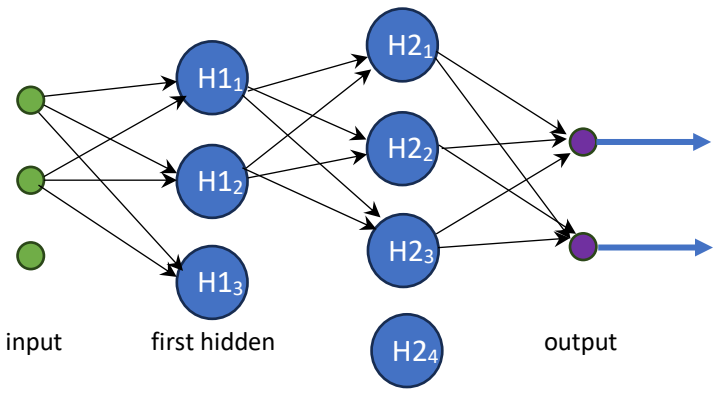


Figure 9.11. Graphic representation of a Feed Forward Artificial Neural Network (FFANN) and its layers.

through an example. Run the “Classifiers” software that comes with the vision library and create four data sets with the following properties:

$$m_1 = (2.0, 2.1) \quad C_1 = \begin{bmatrix} 1.0 & -1.0 \\ -1.0 & 1.0 \end{bmatrix}$$

$$m_2 = (4.0, 4.1) \quad C_2 = \begin{bmatrix} 1.0 & 1.6 \\ 1.6 & 2.0 \end{bmatrix}$$

$$m_3 = (6.0, 10.0) \quad C_3 = \begin{bmatrix} 2.0 & 0.0 \\ 0.0 & 2.0 \end{bmatrix}$$

$$m_4 = (2.5, 6.0) \quad C_4 = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.0 \end{bmatrix}$$

If you entered these parameters correctly, you should get a sample distribution similar to what is shown in Figure 9.12. If you made a mistake entering those parameters don’t worry about it. The “Classifiers” project that comes with the library already has them as a .cvs file. All you have to do is push the “LOAD TRAIN SAMPLES FROM CSV FILES” button. You can save these samples or the ones that you generated as .csv files by pressing the “SAVE TRAIN SAMPLES FROM AS FILES” button. This button saves these samples into the “trainset.csv” file and the membership data in the “membership.csv” file. When written side-by-side, these two files look like:

trainset.csv	membership.csv
2.60586069235107;0.99493639210926	1;0;0;0
3.33106689223987;1.24901873438536	1;0;0;0
1.91671192119737;2.48858088059000	1;0;0;0

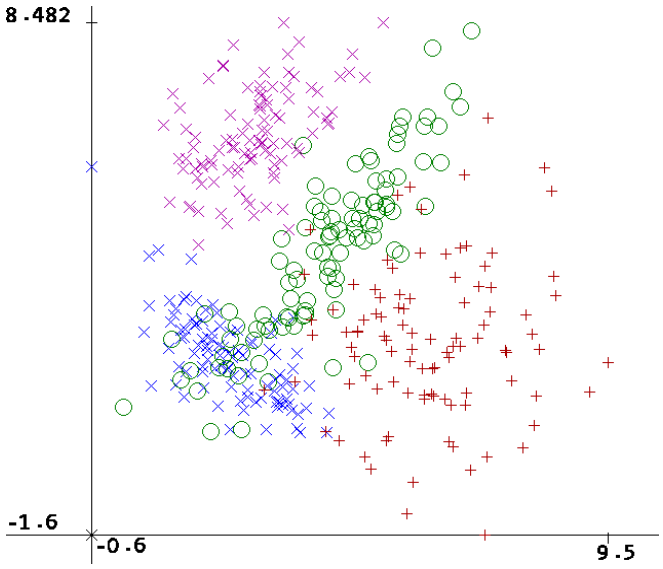


Figure 9.12. Graphic plot of four processes defined by mean and covariance matrices above.

The trainset.csv file contains the x and y coordinates of each sample while the membersip.csv file contains membership data; an indication of which process the corresponding coordinate belongs to. In this example, the membership data contains 4 entries per row (1;0;0;0). This type of membership entry is called “binary” because only the corresponding column that the data’s membership is indicated by “1” and the rest of the columns are “0.” In this case we understand that the first entry in the trainset.csv file, (2.60586069235107; 0.99493639210926) belons to the first process (blue crosses). If it had been a member of the second set (or the process) then the entry in the corresponding row of membersip.csv file would have to be (0;1;0;0).

In ICBYTES terminology, if we know which data belongs to which process by birth, we call them “**membership**.” However, if an unknown data is given to us and we use some sort of classifier to assign (estimate) memberships for those data, then we call that membership data “**labels**.” We use the word “membership” when we are sure, and we use the word “labels” when we estimate.

There are two types of membership/label matrices. The first one is the “binary” type which you already know. The second one is called the “numeric” type:

BINARY	NUMERIC
1;0;0;0	1
0;1;0;0	2
0;0;1;0	3
…	…
…	…

Why do we use two different formats to represent the same data? Obviously, it is more efficient to represent the membership/label data using the numeric format. Unfortunately, ANN outputs cannot generate numeric output. They generate binary outputs and therefore we need to use both formats. Fortunately, the ICBYTES library has functions to convert one format to the other:

```
void BinaryLabelsToNumeric(ICBYTES& inp, ICBYTES& outp, int type);
```

```
void NumericLabelsToBinary (ICBYTES& inp, ICBYTES& outp, int type);
```

The type entry at the end must be variable type like (ICB_SHORT, ICB_INT … etc.)

9.11. Training a FFANN using MATLAB

Once you create the training samples, you need to save them as a .csv file. You can do that simply by calling the following function:

```
ICBYTES path;
path = "trainset.csv";
WriteCSVFile(path, samples);
```

The WriteCSVFile() function will generate a .csv file at any location on your computer and write the contents of “samples” fluid (doesn’t matter if it is a matrix or a vectorlist) into that file. You need to do the same thing for the members. However, first you need to convert them from numeric to binary:

```
ICBYTES binary;
NumericLabelsToBinary(members,binary, ICB_UCHAR);
path = "membership.csv";
WriteCSVFile(path, binary);
```

Now we need the Matlab program to read those .csv files. In order to do that, we must first set the “Current Folder” of Matlab to the folder where we wrote those .csv files. Then, we can type the following commands on Matlab command window:

```
data=readmatrix('trainset.csv');
members=readmatrix('membership.csv');
```

These two commands should load our trainset into the matrix “data,” and the membership info into the “members” matrix. Then we need to create a Feed Forward Neural Network using the command:

```
ann=feedforwardnet(30);
```

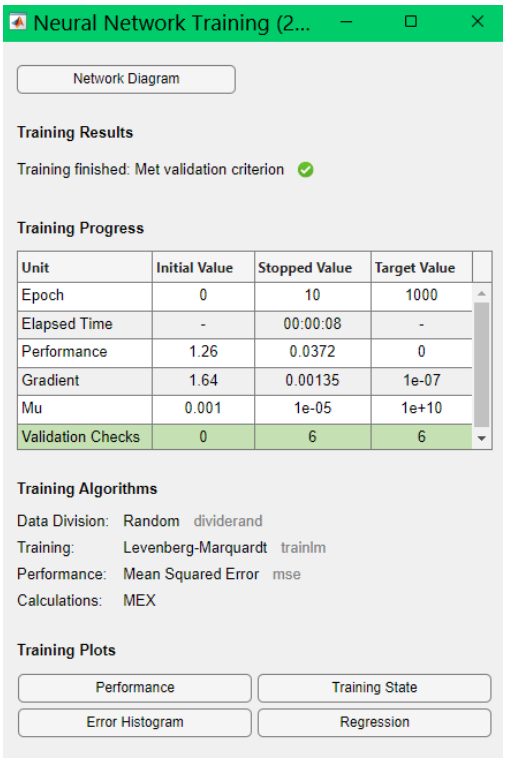
This command creates a FFANN with one hidden layer that has 30 neurons. If we wanted to create two hidden layers with the first one having 7 neurons and the second one having 13 neurons then we would have to type:

```
ann=feedforwardnet([7 13]);
```

Using this notation, you can create FFANNs with any number of hidden layers and neurons. Note that Matlab doesn’t require you to enter the number of inputs or the outputs. That is because, it can figure that out by your training data. Now that everything is set, all we have to do is call the training function by typing:

```
ann=train(ann,data',members');
```

ONCE YOU TYPE THIS COMMAND NOTHING WILL HAPPEN FOR A LONG TIME! DO NOT CLOSE THE MATLAB PROGRAM! EVENTUALLY THE FOLLOWING WINDOW SHOULD APPEAR:



As soon as this window pops up, you can save your Neural Network as a .csv file so that it can be exported back to the ICBYTES library. If you look into the folder of the "Classifiers" project, you shall see a file called "saveANN2ICBYTES.p". That file saves the ANN trained by the Matlab program as a .csv file to be imported by the ICBYTES library. Now you need to type:

```
saveANN2ICBYTES(ann, 'NeuroNET.csv');
```

Check to see that the "NeuroNet.csv" (or whatever the filename you chose) is written on the hard disk. If it is there, congratulations. You successfully exported data from ICBYTES to Matlab, trained a feed forward ANN and you are ready to import it back to ICBYTES library.

9.12. Importing an ANN to ICBYTES

Once an ANN has been exported by some program as a .csv file, it can be imported very easily. The following three lines of code demonstrates how it is done:

```
ICBYTES path, MYANN;
path = "NeuralNET.csv";
LoadFFANN(path, MYANN);
```

The weights and biases of the network which had been written inside the "NeuralNET.csv" file must be imported as a vectorlist. After importing the network, you can use it to classify data by calling the FFANN() function:

```
FFANN(MYANN, data, label);
```

The "label" matrix contains the output of the network. These outputs are the actual outputs of the neurons, therefore, they are floating point (double) numbers between (0.0, 1.0) or (-1.0, 1.0). You need to threshold them to extract the info and the threshold value and methods may differ based on the type of network or training.

Figure 9.13 shows the result of classification using the parameters inside the NeuralNet.csv file. That network has one hidden layer with 30 neurons.

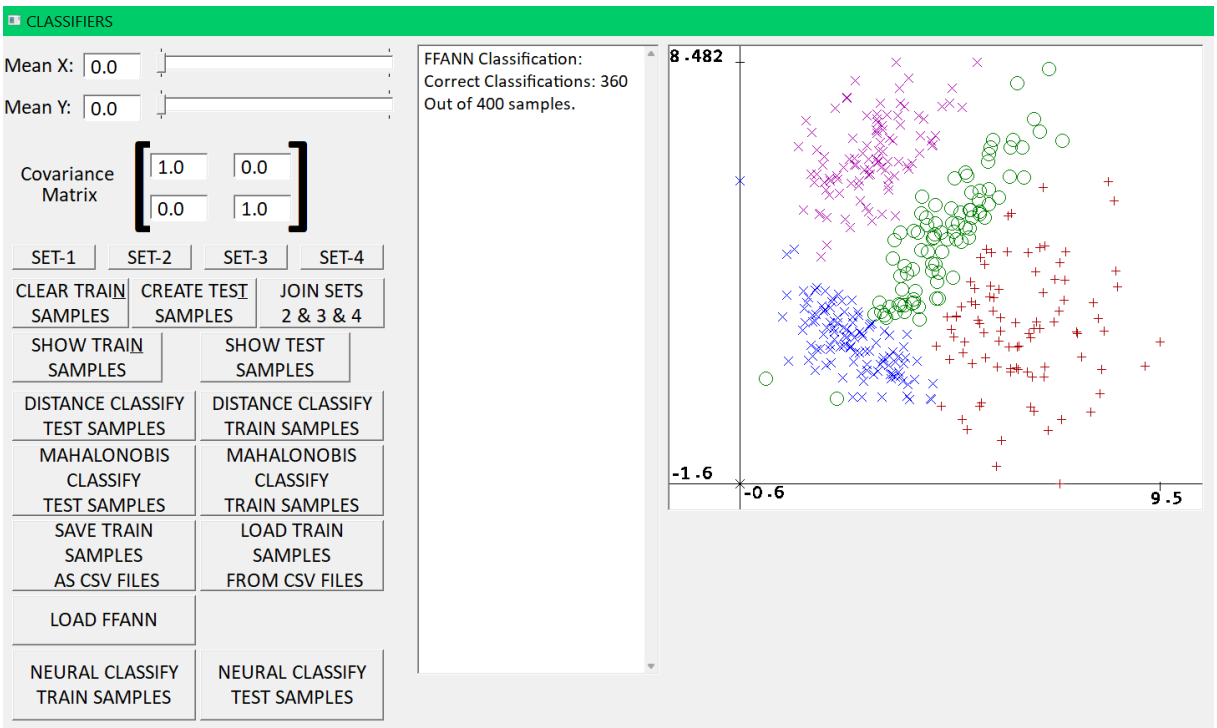


Figure 9.13. Classification results of the data that was shown in Figure 9.12 using the FFANN trained by the Matlab program.

Note that the text window states that 360 samples out of 400 were classified correctly. The function that performs this calculation is the CompareLabels() function. Theis function's first argument is a matrix that contains the membership information. Remember that the term "membership" is used for %100 real information defining the membership of samples which is created during the creation of the samples. If this membership is a result of estimation by a classifier, then we call them "labels." This function's second argument is a matrix that contains those labels. The CompareLabels(members,labels) function returns a positive integer indicating how many of those labels calculated by a classifier have correctly matched with actual membership.

If you press the "DISTANCE CLASSIFY TRAIN SAMPLES" button, you shall see that only 341 of them can be classified correctly and if you press the "MAHALONOBIS CLASSIFY TRAIN SAMPLES" button 354 of them can be classified correctly.

10. Data Transfer & Security

Interconnectivity is a crucial part of every computer system. A robotic system requires continuous communication between processor cards, cameras, sensors and a centralized remote-control station that may be coordinating the robots. It is very important to protect this communication from outsiders for two main reasons:

1- If the communication between these hardware nodes gets compromised, the robot can be hijacked or used by hackers to perform malicious tasks. A very famous example of this is Iran's hijacking of Lockheed Martin RQ-170 Sentinel UAV (belonging to USA) on December 5th 2011. When this plane was flying over the city of Kashmar in Iran, the cyberwarfare unit of Iran took control of the aircraft and safely landed it on an airport. Afterwards, Iran produced reversed engineered versions of that UAV and called it Shahed-171 Simorgh. **This might mean that AES (Rijndael) cipher is NOT SECURE.**

2- Robots gather commercially valuable data while manufacturing goods such as the details of the construction of the goods, the layout of a building or commercially confidential manufacturing systems and methods. Leakage of such crucial data to a competitor may hamper a company's survivability.

On top of these, there is the matter of privacy. Protecting the privacy of individuals in commercial settings such as factories may not be as important as those two reasons listed above, since workers in a factory do not reveal private things unless the robot is a toilet cleaning robot. Nevertheless, a good engineer must always consider every aspect of his/her design not to let any harm befall on individuals.

ALL OF THE ENCRYPTION ALGORITHMS INSIDE THE ICBYTES LIBRARY ARE EITHER **NOT** PATENTED OR THEIR PATENTS HAVE **EXPIRED**. YOU CAN USE THEM FREELY.

10.1. Data Transfer Methods

The ICBYTES library allows many different methods of data transfer. The ones that are implemented so far are the Internet, the serial port and the file transfer through disks. Encryption algorithms that will be introduced in this chapter are compatible with all of those methods and in the future, if other methods are implemented, they will be made compatible as well. Therefore, the user can be assured that the ICBYTES library provides **utmost** security for any and all communications, should the user wish to implement them.

How to combine security features of the ICBYTES library with its data transfer abilities will be demonstrated in section 10.6. after the security concepts are explained.

10.2. Security Strength of Algorithms

There are hundreds of encryption algorithms out there and the user might be wondering why we chose the following few. When choosing an algorithm, we focused on two criteria:

1- Is the algorithm old, famous and still secure?

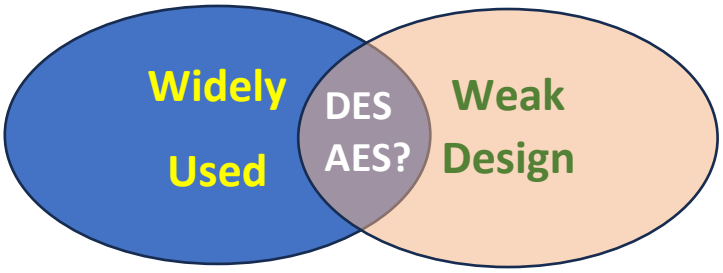
2- Is the algorithm obscure yet secure?

If an algorithm is famous and used widespread, that motivates crypto experts around the world to look for weaknesses. Finding a mathematical weakness in a famous algorithm makes those experts famous as well. One example of that is the DES (**D**ata **E**ncryption **S**tandard) algorithm. That algorithm is the most famous encryption algorithm and was recommended by the American government. Unfortunately, over the years, serious weaknesses have been identified about this algorithm. The weaknesses are so severe that this algorithm, which is originally a 56-bit is now considered 48-bit. That is why people use it in triple-DES format now. On the other hand, if an algorithm is popular, old and no weaknesses has been found despite the attacks of cryptoanalysis experts, then it can be considered very safe.

What is worse about the DES is that the DES algorithm's design allows it to run much faster on dedicated hardware (chips designed particularly for encrypting) than on software (see S-BOX). Overwhelming majority of civilian encryption is performed using software, NOT hardware. Why would a government recommend an encryption algorithm to its citizens that has slow implementation in software but fast implementation in hardware?

It is believed that over the years many governments, organizations and even hacker groups have procured thousands of DES decryption chips to apply brute force attacks on that encryption. In other words, when the American government recommended that algorithm as a standard, it drew a large bullseye on it. What is funny is that the NSA, America's largest government organization whose task is to break encryption, is known to have invested millions on those DES-breaking chips. That makes us think that from the start, they may have recommended this algorithm knowing a weakness about the DES that no one else knows, so that they can eavesdrop on their own citizens. You might think that we are being paranoid, but as this Author's cybersecurity mentor, Dr. Levent Ertaul has always said: "A good cybersecurity expert must be a little paranoid."

That is the main reason why the DES and the AES are not included in the ICBYTES library. Though there have been no serious weaknesses identified about the AES (**A**dvanced **E**ncryption **S**tandard), the fact that the American government made it the **new** standard for its own government agencies and workers made it a huge target for all intelligence agencies and hackers around the world. Remember that when the British broke the Enigma encryption of Germans during the World War-II, for years they didn't tell anyone so that they can keep eavesdropping on Germans. The same thing might have happened to AES already. If a foreign country or hacker group found a way to break AES, you wouldn't know. The fact that Iran was able to hack a top security US drone might be a strong clue that Iran or one of its allies such as Russia or China has already broke AES.



On the other end of this spectrum are obscure but highly praised algorithms such as the HC-256. No one has any good reason to invest millions of dollars to break that algorithm because it is not being widely used in the world. Yet, security experts think that it is a very strong cipher. That is why HC-256 is included in the ICBYTES and algorithms like that will be added in the future.

10.3. The Checksum

Checksum algorithms are low level security algorithms that alert the user when the data has been altered. All of us are using these algorithms without knowing whenever we read from or write to a file on the hard disk. Hard disks and CD/DVD devices all have built in security features to detect whether the files they have on them have been corrupted or not. Consider a magnetic hard drive: Every bit of your file is written as tiny magnetic fluctuations the size of a micron on a disc that is rotating more than 10000 times a minute. The probability of making a mistake is not low. A simple way to find out if a file has been corrupted or not is simply to sum all the bytes in the file and write this sum at the end of the file. For example, if you open the "Notepad" program in windows and type "ABC" and then save that file, you would have actually written the ASCII values of those characters which are:

65 66 67

A B C

So, the computer writes 65+66+67 = 198 at the end of the file. If by accident, the file gets corrupted and reads:

A E C

65 69 67

then the next time you open that file, the hard disk (or the OS) will perform the checksum calculation and find out that the checksum of the file is 201 but the checksum written at the end of the file is 198. Then the computer will warn you that the file is corrupted and it cannot be read.

Of course, the actual checksum algorithms are not that simple. Such a checksum algorithm would fail if the file became ACB or BCA or something like that. The actual ones used on the hard drives are called CRC checksum (Cyclic Redundancy Check) algorithms and they are a family of functions that are much more complicated than simply adding all the byte values. Unfortunately, the more secure an algorithm is, the more complicated and slower (or memory consuming) it becomes. Nevertheless, checksum functions are essential for insuring data integrity during communication and a robotic system has to do a lot of that. Therefore, two different checksum approaches were implemented in the ICBYTES library.

The first one is the CRC-32 algorithm. This algorithm has many versions but the most widely used version is the ISO-HDLC version. This is the version that people usually call simply CRC-32. Other versions have extended names such as CRC-32/ISCSI or CRC-32/BZIP2. If you open the "KRiPTO" project that comes with our library and compile it, you will see that there is a button that says "CRC-32 TEST VEKTOR" on it. If you press that button, it will perform one of the most common vector tests for CRC-32. Test vectors are simple set of input data and the expected output of the algorithm is usually published by the inventor of the algorithm. In our case, they are:

```
Test Vektor (ASCII):    123456789
CRC-32:   cbf43926
```

The function that calculates this checksum is CRC32() function. It is used exactly the same way as the Adler checksum which is explained below.

The 32-bit Adler checksum provides enough reliability for low level security operations while also providing fast processing speeds. Neither CRC-32, nor the Adler checksum are patented therefore you can use them **FREELY**.

In the "KRiPTO" project, you shall see the button that says "ADLER CHECKSUM." That button performs the 32-bit Adler checksum on a file called "Test.txt" which contains bunch of lines that says:

```
ABCDEFGHIJKLMNOPRSTUVWXYZ_0123456789
```

If you press that button, the text window displays:

```
Adler Checksum (Test.txt): efb675ef
```

Now open that file, and switch the places of letters A & B on the top line so that it says:

```
BACDEFGHIJKLMNOPRSTUVWXYZ_0123456789
```

Save the file and press the button again and notice that the checksum has slightly changed:

```
Adler Checksum (Test.txt): efb775ef
```

Proving that even such a small change in the file can be detected by the Adler checksum. Now let's take a look at the function that performs this checksum. But before doing that, **please change the "Test.txt" file back to the original so that it reads "ABCDE…" again; NOT "BACDE…"**

```
void Adlertest()
```

```
{
	ICDEVICE file;
	ICBYTES buffer;
	long long adlermem;
	unsigned adler;
	CreateMatrix(buffer, 512, ICB_CHAR);
	if (!CreateFileReader(file, "Test.txt"))
	{
		ICG_printf("Can't find Test.txt Bulamadım\n");
		return;
	}
	ICG_SetprintfColor(0xff0000);
	while (ReadIntoBuffer(file, buffer))
		adler = Adler(buffer, adlermem);
	if (adler)ICG_printf("Adler Checksum (Test.txt): %x\n", adler);
}
```

We see that the function creates a ICDEVICE named “file,” to access the “Test.txt” file. We also see that it creates a **char** vector with a length of 512 bytes, to be used as a memory buffer. Files that we wish to process may be gigabytes long. We cannot create a RAM buffer to hold the entire file. Therefore, we need to process the file piece by piece, in our case, 512-byte pieces at a time. That is why we create the “buffer.”

After the CreateFileReader() function successfully opens the file for reading, we see that a while loop reads 512 bytes into the buffer and the Adler() function processes it. When the ReadIntoBuffer() function reaches the end of the “Test.txt” file which is 3342 bytes long, it returns 0 and that ends the while loop. If the Adler() function is successful, it returns a value greater than zero and that value is printed on the screen. Note that the Adler() function requires two arguments. The first one is the ICBYTES type “buffer” which holds a 512-byte long vector (array). The second one is “adlermem” which is a 64-bit (**long long**) integer. That integer is used to hold intermediate data to provide continuity for the Adler() function between the processing of 512 byte pieces.

10.4. The HC-256 Stream Cipher

HC-256 is a very powerful stream cipher designed by Hongjun Wu in 2004. It is **NOT PATENTED**. Therefore, we can use it in our projects freely. (Thank you Dr. Hongjun Wu). This encryption algorithm is not only considered very secure but also very fast. A stream cipher encrypts data byte-by-byte, therefore, the resultant ciphertext is exactly the same size with the original plain text. On the

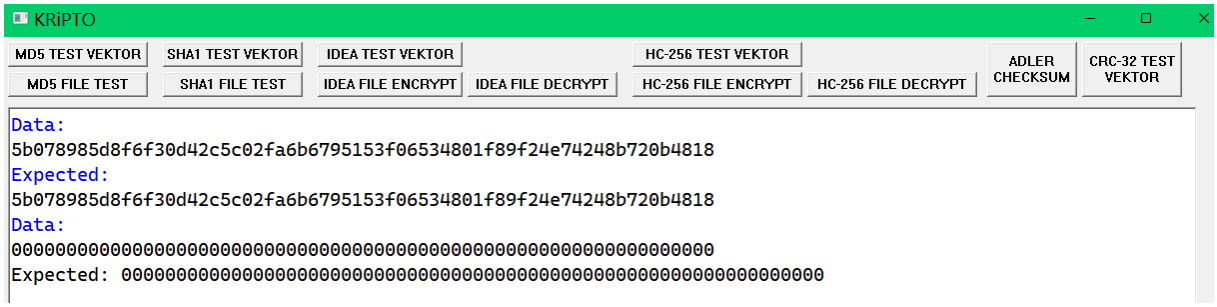


Figure 10.1. The screenshot of the KRiPTO program displaying successful calculations of a test vector using HC-256.

other hand, as we will see in the next section, block ciphers process plain text in blocks of 8 bytes and as a result, change the size of the file.

Encryption algorithms always come with test vectors. A test vector is a short plain text and the result of its encryption using a specific key. Programmers use these test vectors to prove that their implementation works exactly as the original design that the inventor of that encryption algorithm intended. Figure 10.1 shows the screenshot of the KRiPTO program when the “HC-256 TEST VEKTOR” function has been pressed. The test vector is 32 zeros and the second test vector is decryption of it back to plain text. If you press the “HC-256 FILE ENCRYPT” button, it will encrypt the “Test.txt” file. If you open that file using notepad after encryption, you shall see bunch of garbled data as shown in Figure 10.2.

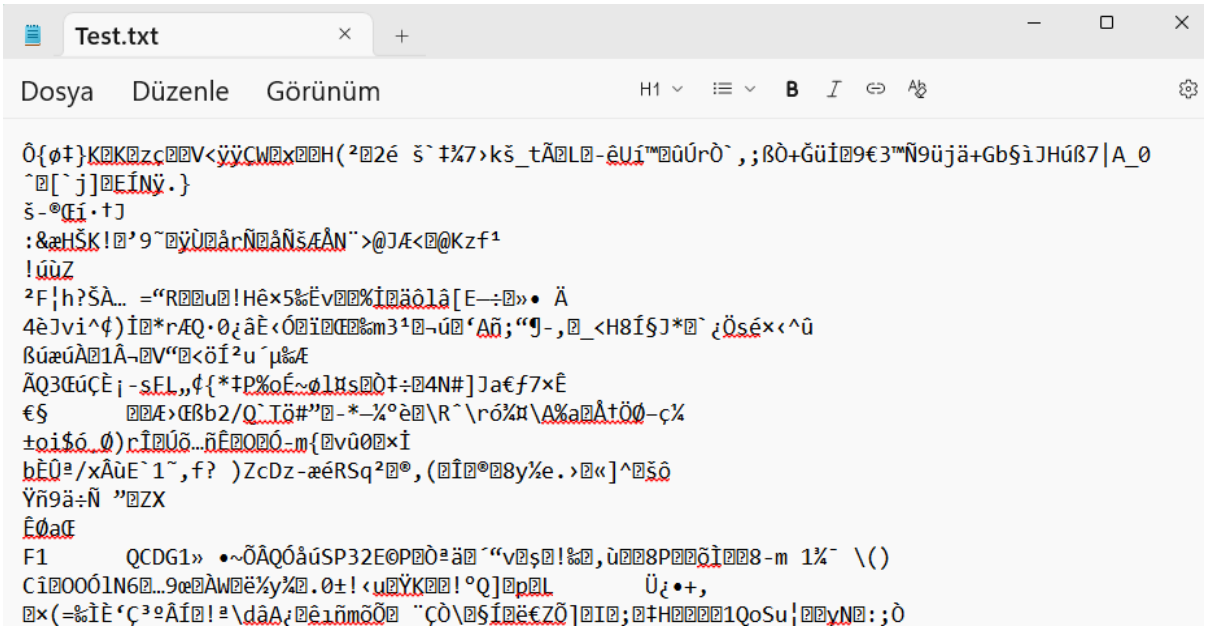


Figure 10.2. The Test.txt file displayed by Notepad program after being encrypted using the HC-256 algorithm.

After opening the file using the Notepad, close it without making any changes. Then press the “HC-256 FILE DECRYPT” button which will turn the encrypted Test.txt file back to its original form. When you open it using the Notepad again, you should see many lines of plain text:

```
ABCDEFGHIJKLMNOPRSTUVWXYZ_0123456789
ABCDEFGHIJKLMNOPRSTUVWXYZ_0123456789
ABCDEFGHIJKLMNOPRSTUVWXYZ_0123456789
...
```

Now let’s take a look at the code that performs the HC-256 file encryption:

```
void HC256fileencrypt()
{
  ICBYTES data, key, iv, filename;
  IncreasingMatrix(key, 32, ICB_UCHAR);
  IncreasingMatrix(iv, 32, ICB_UCHAR);
  key *= 1273; iv *= 3177;
  filename = "Test.txt";
  ICDEVICE file;
  if (!CreateFileDevice(file, (const char*)&filename.B_[1][1]))
  {
      ICG_printf("Deneme.txt Can't find / Bulamadım\n");
      return;
  }
  CreateMatrix(data, 256, ICB_CHAR);
  while (ReadIntoBuffer(file, data))
  {
      HC_256_ENCRYPT(data, key, iv);
      WriteBuffer(file, data);
  }
  ICG_printf("Test.txt encrypted using HC256. (Test.txt HC256
kullanılarak şifrelendi.)\n");
  CloseDevice(file);
}
```

First of all, we need to create the file buffer, which in this case is called “data.” Then the other essential fluids we will need are the ‘key’ which holds the encryption/decryption key (kind of like a password) and there is another fluid ‘iv’ which is like another key but it is actually used to obfuscate plain text before it is encrypted.

The ’iv’ is an important part of strong encryption. Most of the time, a hacker can guess parts of your plain text. For example, if your plain text is a letter, you usually start with “Dear Sir/Madam …” or if you are encrypting a C++ file the first line is usually “#include<…” Being able to guess such parts of plain text helps crypto breakers to decipher your encrypted files. To prevent that, what you can

do is garble the plain text by exoring the bytes with random data or something similar to that. Since the topic of this book is not cryptography, we will not discuss it further. What the user needs to know is that if he/she wants protect data from even secret government agencies, that is the proper step to take.

If we go back to the code above, we see that we are using the IncreasingMatrix() function to create the 'key' and the 'iv; which fills these vectors with [0, 1, 2, 3, 4 …]. Certainly, this is not a good key/iv. A good key/iv should be very unpredictable however, our goal here is to explain the students how to use these cryptography functions quickly. Then you see the lines:

```
key *= 1273; iv *= 3177;
```

Which means that every byte inside the 'key' vector is multiplied by 1273. Keep in mind that these fluids are of type (unsigned char). An unsigned char can hold numbers between 0 and 255. So, if you multiply '1' with 1273 and try to put it in an unsigned char variable, you can only fit the least significant bits which equals:

$$1273(mod\ 256) = 249$$

Therefore, the 'key' vector will hold: [0, 249, 242, 235, 228, 221, …] which is a bit more unpredictable. On the other hand, the 'iv' is multiplied by 3177 yielding the following sequence inside it: [0, 105, 210, 59, 164, 13, 118, …]. Again, keep in mind that these are not proper/secure ways of generating keys. In real life do NOT use these methods.

The next step is to create a file device. In the previous section, for the Adler checksum, we had used the CreateFileReader() function because all we wanted was to read the files, not to alter them. But for encryption, we need to both read and write onto the file. That is why this time we used CreateFileDevice() function which will allow us to read from and/or write on the file.

After successfully opening a file device, the next thing we do is allocate space for our file buffer 'data.' We chose to allocate 256 bytes for this buffer by calling the following function:

```
CreateMatrix(data, 256, ICB_CHAR);
```

It does not matter whether you create char or unsigned char buffer for file processing. You may want to create larger buffers if you have large RAM because that will speed up encryption/decryption process considerably. The next step is to create a while loop to process the file 256 byte at a time and write the encrypted part exactly on top of the same place in the file:

```
while (ReadIntoBuffer(file, data))
{
    HC_256_ENCRYPT(data, key, iv);
    WriteBuffer(file, data);
}
```

Once this loop is done, the encryption is complete. The decryption code is exactly the same, the only difference is we use HC_256_DECRYPT() function.

10.5. The IDEA Block Cipher

IDEA (International Data Encryption Algorithm) is an old but very strong 128-bit block cipher. It was used by financial institutions in Europe for information exchange and since its design in 1991, no serious attacks have been developed against it, provided that the user does not use weak keys. It is one of the few algorithms that employ modular multiplication. Over the years many security experts praised it for its high security. The patent of this algorithm has expired in every country making it **FREE**. Using an old algorithm that has proven its strength is a lot more secure than using a new but untested algorithm.

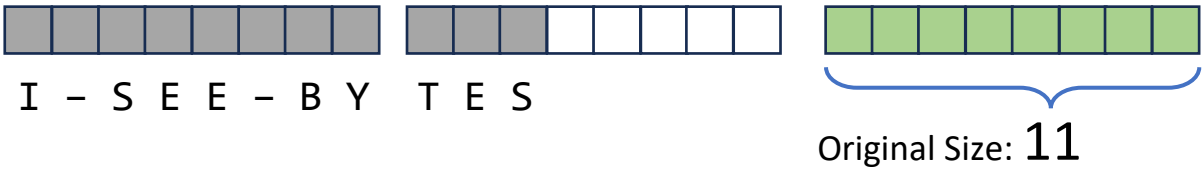


Figure 10.3. How block ciphers such as the IDEA force the size of the encrypted file to expand.

Block ciphers encrypt data 8 bytes (64 bits) at a time allowing the bits to shuffle at a larger range. That is why they are considered more secure compared to the stream ciphers. However, as seen in Figure 10.3, that forces the size of the plain text to change. Imagine that we opened the Notepad program and wrote "I-SEE-BYTES" in it and saved. That file would be 11 bytes long. Unfortunately, a block cipher requires blocks of 8 bytes to operate. Therefore, we need to extend the file another 5 bytes to make it 16 bytes (integer multiple of 8) by padding random data, for the IDEA to process all of the file. This means that, when we decipher this file, we need to know how many bytes the original file was, so that we can shrink it to the original size. That means adding another 8 bytes of length information somewhere (usually at the end) in the encrypted file.

As with the other cryptographic algorithms, the authors of IDEA provided us with test vectors so that we can ensure that our implementation works exactly as they described. If you press the "IDEA TEST VEKTOR" button on the KRiPTO

project, you shall see that one of the test vectors is encrypted and then decrypted:

```
Data:
UCHAR (8 × 1)
0 0 0 1 0 2 0 3

Key:
UCHAR (16 × 1)
0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8

Encryption result:        11fbed2b01986de5
Expected Result:          11fbed2b01986de5
Decryption result:
UCHAR (8 × 1)
0 0 0 1 0 2 0 3
```

If you look at the code of IDEA file encryption inside the KRiPTO project, you will see that it is long and complicated because this function appends the file length itself and creates a buffer. There is an important issue about the buffer size:

!!! IDEA REQUIRES BUFFER SIZES THAT ARE INTEGER MULTIPLE OF 8 !!!

If you create a buffer that is NOT an integer multiple of 8 then the encryption/decryption may be faulty. A safer way to encrypt/decrypt files using IDEA is to use built in function as done in the ideafiledecrypt() function which runs when you press the “IDEA FILE DECRYPT” button as shown below:

```
void ideafiledecrypt()
{
      ICBYTES filename, key;
      filename = "Test.txt";
      key = "1234567890abcdef";
      long long filelen = IDEA_DecryptFile(filename, key);
      if (filelen)
            ICG_printf("Test.txt decrypted using IDEA. (Test.txt IDEA
kullanılarak şifresi açıldı.) (%d Byte)\n", filelen);
}
```

As you can see this function calls IDEA_DecryptFile() function to decrypt the file. All you need to do is input the filename / path and the encryption /decryption key to the function. This function has actually three input parameters but the last one is optional. If you look at the definition of that function inside the “ic_cipher.h” file:

```
long long IDEA_DecryptFile(ICBYTES& filename, ICBYTES& key, int bufferblocks = 4);
```

The third optional argument is the number of buffer blocks. Each block is 512 bytes long and therefore the default size of the buffer is 2048 bytes (2K). You can alter this option based on your RAM availability and speed requirement. Same option exists for the encryption as well:

```
long long IDEA_EncryptFile(ICBYTES& filename, ICBYTES& key, int bufferblocks = 4);
```

Both return the size of the file after the encryption /decryption is complete.

10.6. Cryptographic Hash Functions

Cryptographic hash functions are very similar to checksum functions but they are much more sensitive to change and the checksum they calculate is much longer. As a result of these properties, the checksum they generate resembles a signature for that document because it is almost unique. Creating two documents with the same signature, which is also called a "digest," is very hard.

The ICBYTES library includes two different cryptographic hash algorithms: MD5 and SHA1. These algorithms are very old and naturally **their patents have expired long ago**. These algorithms are not considered very safe anymore and there are much better hash algorithms in the literature. Serious cryptographic architectures abandoned these algorithms long ago. However, they have become the default digest generators for the ordinary internet folk. You may have noticed that many web sites publish the MD5 and SHA1 digests of files that they make available for download. The reason for that is to prevent hackers from inserting viruses or malicious code into their software. You may think that, if these hash functions are not very safe why are they still being used by everyone? The first reason for that is, when everyone starts doing the same thing, it becomes a habit that is hard to break. The second reason is, while MD5 and SHA1 are not super secure when used alone, they become very secure when used together. Fooling one algorithm is very difficult. Fooling both of them at the same time? Not a chance.

Finally, for our purposes, the security level of these algorithms is already overkill. We aspire to design a robot that is reliable and difficult to hack. We are not designing an international finance exchange platform or a computer communication system for the MIT (Turkish Intelligence Agency). Besides, these algorithms not only slow down communication because of the considerable processing power they use, but also because of the number of bytes that they generate. For example, MD5 generates a digest that is 16 bytes long and SHA-1

generates a digest that is 20 bytes. When a robot communicates with its peripherals, most of the time it sends and receives a few dozen bytes. How many bytes does a laser distance sensor send at a time? Four? If you use 16 bytes to ensure the integrity of 4 bytes that means you are increasing the data fivefold. That's not good engineering. It probably means that you are either stupid or obsessed with security and paranoid of hackers. For our purposes, CRC-32 algorithm (only 4 bytes long) is adequate for ensuring the integrity of most of the device communications.

Besides, if a cryptographic system already has strong encryption algorithms like IDEA and HC-256, data integrity breaches are not easy to create by hackers. For us, the most useful application of these hash algorithms is for creating keys that IDEA and HC-256 needs. Because of the random nature of the digests created by MD-5 and SHA-1, **they are recommended for creating keys from pass phrases**. That is where we will need them the most.

If you read the code inside the KRiPTO project, you will see that test vectors are provided for both of the hash functions. The file processing code is very similar to the one we used for the Adler checksum. The only difference is when you press the "MD5 FILE TEST" button, file selection dialog box comes and allows you to select any file on your computer. This way, you can use the KRiPTO software to verify the files that you download over the internet. If you chose the" inmoth.csv" file inside the "MEDIA" directory that comes with the ICBYTES library files, you should get 16 bytes in hexadecimal:

```
C:\2- REL_VISION\MEDIA\inmoth.csv
f0bebf2c8be9f226132799d6c3092bcf
```

for the MD5 digest. If it becomes very necessary, the SHA-2 and the SHA-3 algorithms will be included in the future versions of ICBYTES.

10.7. Data Transfer

ICBYTES library has numerous different ways for data transfer. The most common, fast and practical way of transferring data between computers is the internet. The TCP/IP protocol has become so common that nowadays even the tiniest microcontrollers have the ability for ethernet communication. In this example we shall demonstrate how easy it is to establish communication through the internet using the ICBYTES. For this example, we will use the self IP address of the computer, 127.0.0.1.

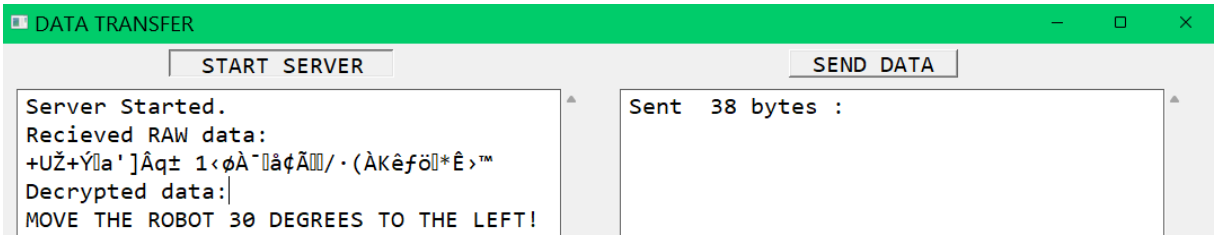


Figure 10.4. The screenshot of the project "Data_Transfer" that comes with the ICBYTES Robot Vision Library.

If you open the "Data_Transfer" project that comes with the ICBYTES library and run it, the following window (Figure 10.4) will appear. You need to press the "START SERVER" toggle button first. Then, every time you press the "SEND DATA" button, the client will send 38 bytes of data which contains encrypted version of the statement: MOVE THE ROBOT 30 DEGREES TO THE LEFT!

In the fifth chapter, we introduced the device used for network communication. Using a server and a client device, we created a server and a client on the same program and had them transfer data between each other. In this chapter, we shall do the same but encrypt the data first.

The simplicity of use and superiority of the I-See-Bytes library is that there is only one data type, the ICBYTES, and one device type, ICDEVICE, allowing seamless interoperability throughout the library.

When you first run the "Data_Transfer" project, you should first press the "START SERVER" toggle button. The function that is called by this button is shown below:

```
void Server(void* p)
{
	ICDEVICE server;
	ICBYTES data, key, iv;
	IncreasingMatrix(key, 32, ICB_UCHAR);//key for decryption
	IncreasingMatrix(iv, 32, ICB_UCHAR);
	ICG_printf(MLE1,"Server Started.\n");
	while (1)
	{
		if (!CreateTCPServer(server, 27015)) break;
		int t = RecvData(server, data);
		if (t <0)break;
		ICG_printf(MLE1,"Received % d bytes : \n", t);
		data.C_[1][t+1] = 0;
		ICG_printf(MLE1, "Received RAW data:\n");
		Print(MLE1, data);
		HC_256_DECRYPT(data, key, iv);
		data.C_[1][t+1] = 0;
```

```
			ICG_printf(MLE1, "Decrypted data:\n");
			Print(MLE1, data);
			CloseDevice(server);
		}
		ICG_printf(MLE1, "Server CLOSED!");
}
```

First, we create a device for our server communication, then we create three fluids that will hold the data, the key for decryption and the IV. Then we fill out the key and the IV with increasing integers. You should use passwords that are much more unpredictable. Then we start an infinite loop.

Inside the loop we create a server device at port 27015 and if that is successful, we start waiting for someone to connect to our server and send us some data. Hopefully, that someone will be our client program. If we receive any data, we display that raw data on the screen, then we decrypt it using the HC-256 encryption algorithm. If everything goes right, we should have the message sent to us by the client. Then we close the server. But because we are inside an infinite loop, the program starts a new server and waits for another client to connect.

In the meantime, if you press the "SEND DATA" button, the following function is called:

```
void Client()
{
		ICDEVICE clnt;
		ICBYTES data, key, iv;
		IncreasingMatrix(key, 32, ICB_UCHAR);//key for encryption
		IncreasingMatrix(iv, 32, ICB_UCHAR);
		data = "MOVE THE ROBOT 30 DEGREES TO THE LEFT!";
		HC_256_ENCRYPT(data, key, iv);
		if (CreateTCPClient(clnt, "127.0.0.1", 27015))
		{
			int t = SendData(clnt, data);
			if(t>0) ICG_printf(MLE2, "Sent % d bytes : \n", t);
		}
		CloseDevice(clnt);
}
```

The first part of this function is almost the same as the previous one. The first difference is, in this function, we fill the “data” fluid with the following characters:

```
data = "MOVE THE ROBOT 30 DEGREES TO THE LEFT!";
```

Then we encrypt the data using the same increasing integers as key, IV, and the HC-256 encryption algorithm. When encrypted, the above statement turns into these garbled characters shown in Figure 10.4:

+UŽ+Ẏ[]a']Âq± 1‹øÀ¯å¢Ã————————————————————/·(ÀKêƒö*Ê›™

The next step is to initiate a client connection to a server whose address is 127.0.0.1, which is this computer, and the port is 27015. If this connection is successful, in other words, if the CreateTCPClient() function returns true, then we send the encrypted data to the server. We printout the number of bytes sent to the server and then we close the connection and end the function.

As you can see, the I-See-Bytes library is so simple that, transferring data with encryption /decryption adds only one extra step.

10.8. Transferring Data Through Disks

Although not as fast as the ethernet, most of the time, transferring data using a USB flash drive or an external hard disk is more practical. The ICBYTES library provides many methods for that but probably the most universal way of doing that is to use CSV (Comma Separated Values) files. These files can be opened by any data processing program including Excel and Matlab. In the 9th chapter we used these files to transfer data to; and Neural Net parameters from the Matlab program. In this section, we shall learn the CSV file reading and writing capabilities of the ICBYTES library in more detail.

Open the “Blob_Ground” project that comes with our library and run it. If you press the "GROUND TRUTH" button and select the “inmoth.jpg” file from the “MEDIA” directory, you should see the application window that is shown Figure 10.5. In the general field of Computer / Robot Vision, when algorithms are being developed, their success rate must be continuously measured. In order to do that we need reliable data with which we can compare the results of the algorithms that we design.

In this example, assume that you are developing a code that can detect moth eggs. In order to measure the success rate, you will need a data file that contains the peripheral data of those eggs. That data is manually created by humans painstakingly so that an algorithm developer like yourself can measure the success rate of his/her algorithm. The file that contains the coordinates of the peripherals of these eggs is “inmoth.csv” and is automatically loaded when you load the “inmoth.jpg” file. Such files are generally called “The Ground Truth.”

Figure 10.5. The screenshot of the project "BLOB_GROUND" that comes with the ICBYTES Robot Vision Library.

If you look at the Ground() function, the function that was called when you pressed the "GROUND TRUTH" button, after opening the inmoth.jpg file, the file name extension is being changed to .csv and the using the following function, the inmoth.csv file is being loaded:

```
ReadCSVFile(path, features, data,trans);
```

The "path" contains the filename and the path of the .csv file. Actual data inside the file is transferred into a data, which is a vector list, the labels at the beginning of the CSV file is loaded into another vector list, features.

If you open the inmoth.csv file using the Notepad, you will see the following:

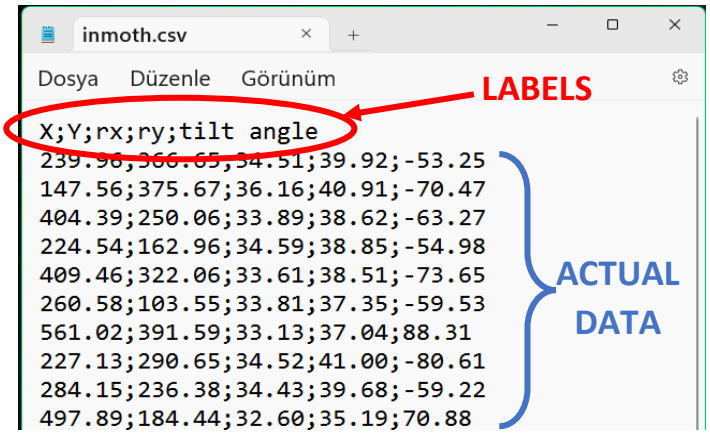


Figure 10.6. The inmoth.csv file when opened using the Notepad program.

The first line of a CSV file usually contains the labels of each column. In our case, it says that the first and the second column contains x and y coordinates of the center of the ellipses. The third column rx is the horizontal and the fourth column, ry, is the vertical radius, and the tilt angle is the last column. We can use this data with the TiltedEllipseArc() function as follows to draw those ellipses:

```
TiltedEllipseArc(RGB, x, y, rx, ry, angle, color);
```

These label names are written inside the “features” fluid. If you print that fluid which actually contains a char type vector list, you will see the following:

1- X
2- Y
3- rx
4- ry
5- tilt angle

Finally, there is also the “trans” fluid which was used as a input to the ReadCSVFile() function. Sometimes CSV files may contain names instead of values. For example, one column may consist of days of the week such as Monday, Tuesday, etc. Then every Monday will be replaced by 1.0, every Tuesday will be replaced by 2.0 and “trans” is the” translation” vector list indicating which word is represented by which number in the “data”. All of the apply for WriteCSVFile() function as well. The “data” must be a matrix.

11. Video Processing

In Robot Vision, a great deal of processing is performed on data captured by cameras. These cameras are usually USB or IP (network) cameras. However, during the development and testing of robot vision algorithms, it is impractical to use the actual robot. It might even be dangerous. That is why, most of the robotic systems are "simulated" using the stored test data during the research and development stage. Camera data is usually stored as images or videos. In this chapter we shall learn the intrinsic details about the data produced by cameras and how to process videos.

11.1. Camera Image Formats

In Chapter 3 we learned about how color images are both stored and displayed as RGB values. That is because the human eye has light receptors for only red, green and blue. However, the human eye does not have balanced number of receptors for each of these colors. Human retina has much more green receptors compared to the red and blue ones. This means that, when we look at a scene, our retinas collect more information from the green color than the other ones. Most of the cameras adapt to this condition, and just like the human retina they have twice the number of green receptors compared to the blue and red ones. As a result, they can't send the data as one RGB value per pixel, because they don't have that information. Instead, most of the cameras use YUV format to send data. In that format, "Y" stands for illumination (brightness), and it mostly consists of green pixel information. Remember the function Luma(.) in ICBYTES? That function is used for calculating Y from RGB images.

The other two components, which are called "chroma" carry the majority of the color information. However, because the chroma information is produced by red and blue sensors in the camera, and because their number is half of that

of the green sensors, the size of the U and V components are usually half, or sometimes even quarter of the Y component. This also allows YUV videos to consume much less space than the RGB videos.

As we all know, most of the video compression formats such as the MPEG-4 cause considerable loss of image information. In the fields of Machine and Robot Vision, such loss of information is usually unacceptable. We prefer to use **un**compressed (raw) image data. (Computer Vision guys work with whatever they can get.) The most common raw image format is the YUV format. That is why almost all of the USB cameras have an option to send data in that format. As a matter of fact, when someone talks about the raw video, they are talking about YUV videos %90 of the time. The second most widely used raw video format is the Motion JPEG (MJPG) format. If you look at the image formats supported by the web cam mentioned in section 5.7, all of the images that the camera sends are YUV or JPEG:

```
1280x720 30fps NV12
1280x720 30fps MJPG
640x480 30fps NV12
640x480 30fps MJPG
640x360 30fps NV12
640x360 30fps MJPG
1280x720 10fps YUY2
640x480 30fps YUY2
640x360 30fps YUY2
```

In the above list, YUY2 means YUV format with U and V are half the size of Y in the horizontal direction. NV12 is another YUV format which we will not describe in detail. As you can see, for the highest resolutions and frame rates, higher compression rate formats such as NV12 and MJPG are preferred because the USB 2.1 cannot transfer raw image data at such high rates.

You might be wondering how we got RGB images from the USB web cameras in chapter 5. That is because either the Windows operating system or the ICBYTES library converts these YUV images to RGB in the background. However, receiving the Y component is very advantageous sometimes because majority of the Robot Vision algorithms work on the Y component. Consider filtering and edge detection. We had to convert RGB to Y using the Luma(.) function.

11.2. Raw Video Files

One of the standard forms of raw image video format is YUV4MPEG2. The extension of video files encoded with this format is ".y4m". Most of the research that use video files require uncompressed image frames and that is one of the reasons why in academia, test data is shared using this format. Many of these files can be downloaded from www.xiph.org.

The ICBYTES library has a device that decodes this type of media. You can simply initiate this device using the following code:

```
ICDEVICE video;
ICBYTES path;
CreateY4MDecoder(video, path);
```

As usual, the "path" vector includes the path and the name of the video file. If the file is a correct YUV4MPEG2 file, then the above function will return true. You can query the frame size or frame rate or you can simply start reading the YUV components of each frame.

The ICBYTES robot vision library comes with an example project on how to use this decoder. You should first download one of the videos from xiph.org before running the project. In this example, we will use "tennis_sif.y4m" file. If

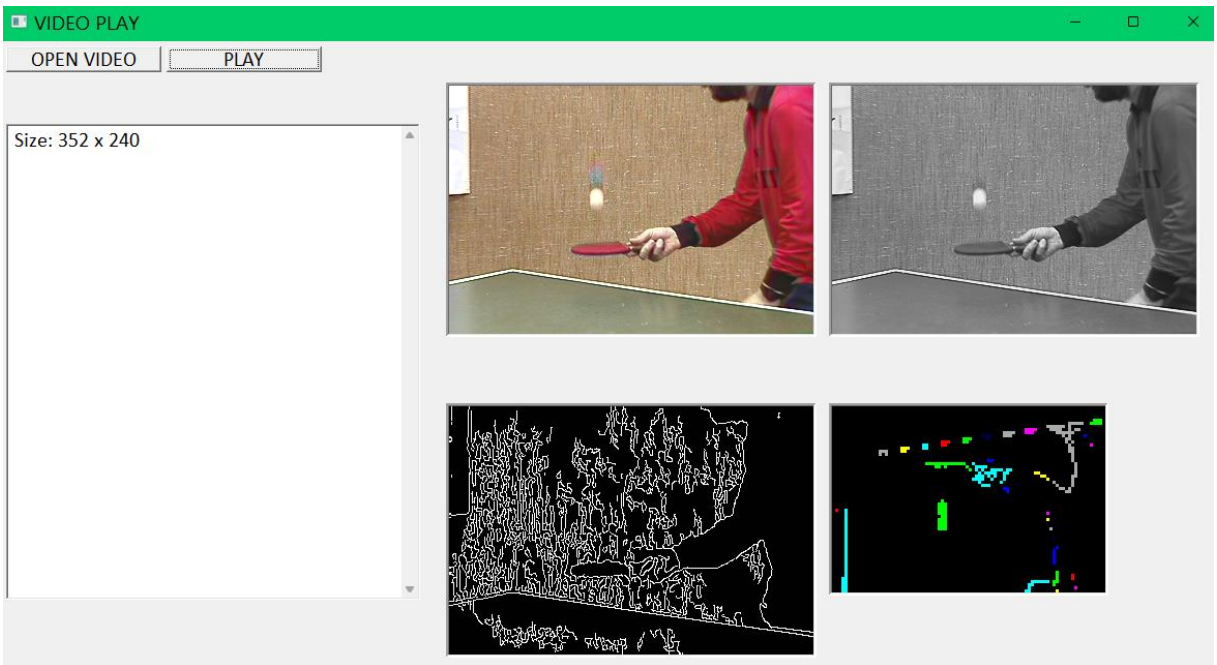


Figure 11.1. The screenshot of the "Video" project that comes with the ICBYTES library.

you run the project and open that video file, you should get the following screenshot (Figure 11.1).

As seen in Figure 11.1, when the video file is opened using the "OPEN VIDEO" button, The program shows four different videos simultaneously. The first one on the upper left is the RGB version, next to that one is the Y component, the one on the bottom left is the edge detected version and the bottom right one shows the moving objects (frame difference) using different colors. But that one is upside down. The message window indicates that the resolution of the video is 352×240 pixels.

The "Video" project has a fundamental difference compared to the other projects. It uses **multithreaded programming techniques**. Before diving into the code of this project, a brief introduction must be given for those who do not know them.

11.3. Multithreaded Programming

Multithreaded programming is the methods and techniques of performing multiple tasks simultaneously inside the same program. These are essentially separate functions that are running parallel to each other. As shown in figure 11.2-a, a conventional (single threaded) program starts with the main(.) function and performs the instructions inside it one by one. Once the instructions finish, so does the program. On the other hand, a multithreaded program (Figure 11.2-b) starts with the main(.) as well, but at some point, the main function calls (spawns) another function that literally runs at the same time with the main(.).

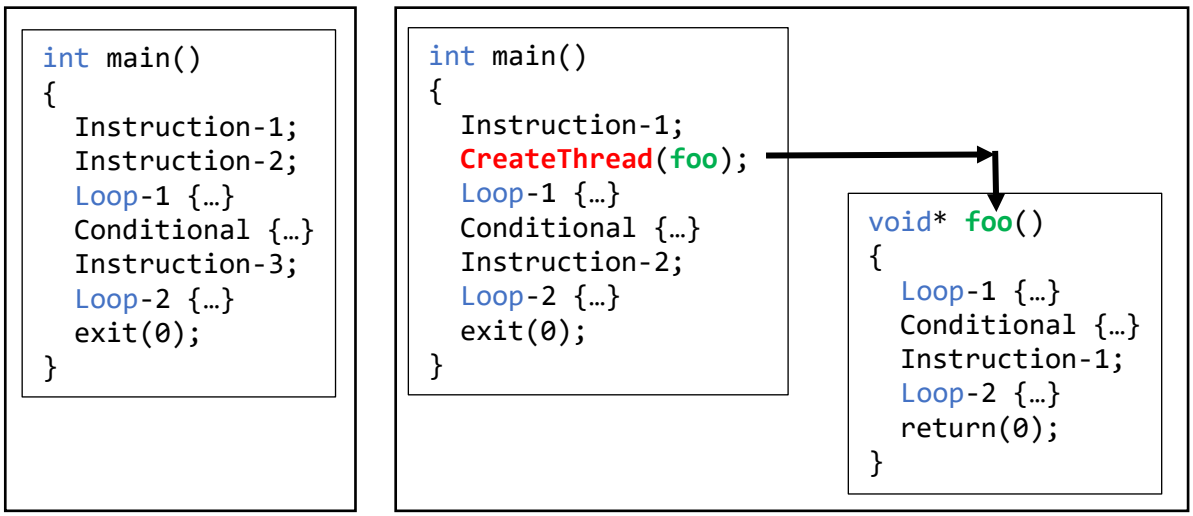


Figure 11.2. (a) A single threaded conventional program. (b) A multithreaded program with foo(.) running at the same time with main(.).

Normally, when main(.) calls a function, it doesn't continue executing the next instruction until that function returns. But in this case, main(.) calls foo(.) and continues executing the next instruction without waiting for foo(.) to return. They start running concurrently. When foo(.) finishes its instructions, it terminates itself without ever telling the main(.) (unless we want it to tell).

The main thread and its child can share the same memory areas if they want to. Any variable defined in the main(.) is private to it and just like any other (normal) function, foo(.) cannot see those variables unless the main sends them to it as arguments. On the other hand, a global variable can be accessed by both of them.

In order to coordinate the behavior of threads, the operating system provides with synchronization tools. There are several of them but since the topic of this book is not concurrent programming (that is the name of the general field), we shall not talk about them. The only one that we will use is the mutex. Mutexes are like tokens. Whichever thread owns the mutex can continue running and the others will have to wait until that thread is done whatever it's doing and releases the mutex. In the "Video" project, we use a mutex to pause and resume the video.

As we mentioned before, a windows GUI needs to process dozens of messages every second sent to it by the operating system. When you move the mouse on top of its window area, messages are sent. When you move the window, messages are sent. When you minimize, maximize or exit the program, messages are sent and if the program can't process these messages because it is busy doing something else then the program **FREEZES**.

We all have experienced windows programs that become unresponsive. That happens because the main thread is busy doing something and can't respond to the messages being sent to it by your mouse (actually it's the OS sending them). You can test this by writing a simple program. Write a function that calls Sleep(10000); command when you press a button. That command does nothing for 10 seconds. Or write a do-nothing-loop that lasts 10 seconds when you press a button. You will see that the program freezes for 10 seconds.

If you need to play a video for minutes or even hours, how are you going to prevent the program from freezing? The answer is by using threads. If you do your long-lasting task in the child thread and let the main thread keep processing the messages sent by Windows OS, then you can have your cake and eat it at the

same time. In the ICBYTES library, the Toggle Button does exactly that. When you create a Toggle Button, the function you attach to it is called as a different thread. That is why its function can keep running without making your program freeze. In the Video project, we will see the mechanism behind the Toggle Button.

11.4. The Video Project

Now let's take a look at the code of the Video project. When we press the "OPEN VIDEO" file, it calls the following function:

```
void VideoPlay()
{
  DWORD dw;
  ICG_ClearText(MLE);
  pause = CreateMutex(NULL,FALSE,  NULL);
  CreateThread(NULL, 0,(LPTHREAD_START_ROUTINE)VideoPlayer,NULL,0, &dw);
}
```

This function clears the text screen and the creates a mutex called "pause" and a thread. The function that runs as a thread is called VideoPlayer:

```
void* VideoPlayer(void* v)
{
	ICDEVICE video;
	ICBYTES path;
	if (NULL == OpenFileMenu(path, ""))return NULL;
	if (CreateY4MDecoder(video, path) != 0) return NULL;
	float frame_rate = GetFrameRate(video);
```

When you reach this point, you opened the video file and created a device called "video" which is capable of decoding YUV4MPEG2 movies. Next, we define the low-pass filter parameters that we will use before the Canny edge detector, and all of the other ICBYTES images and matrices:

```
	//Low-pass filter coefficients
	ICBYTES filtre = { 0.1,0.1,0.4,0.2,0.1 };
	ICBYTES Y, U, V, RGB, Q1, Q2, R, Mag, seg, map, colseg, edge, filtered;
```

The following is the tricky part. To find the frame difference we need to subtract the previous frame (Q1) from the current frame (Q2). But when the next frame comes, current frame becomes the previous frame. We don't want to copy the current frame into the previous frame because that consumes a lot of time and processing power. Instead, we create references for these frames and swap those references (using another reference called temp). That is why we create the following ICBYTES references:

```
	ICBYTES& previous = Q1,  &next = Q2,  &temp=Q1;
```

```
CaptureY(video, Y);
ICG_printf(MLE, "Size: %d x %d", Y.X(), Y.Y());
```

After capturing the first Y component (YUV), we down sample 4×4 (16) pixels into one pixel because we don't care about individual pixels while calculating the frame difference. That is why we use Quart(.). Note that the down sampled version is put in "previous." Then we create a matrix (R) that will hold the frame difference of next minus previous:

```
Quart(Y, previous);
CaptureUV(video, U, V);
CreateMatrix(R, previous.X(), previous.Y(), ICB_UCHAR);
DWORD waitforpause;
```

This is where the actual loop for processing and playing the video starts. As long as we can read a Y frame from the video, we continue looping. But wait! As soon as we enter into the loop we encounter a weird function called WaitForSingObject(pause,…). If you remember, pause is the name of the MUTEX that we created in the previous function. It is a global handle. As a result, we know that we can't continue after this point unless we own the mutex. The good news is, when we are stuck here, our thread consumes zero processing power because the operating system suspends it. Waiting for a mutex is not like creating a do-nothing-loop that consumes huge amount of processing power. Instead, the OS channelizes the CPU to other tasks.

```
while (CaptureY(video, Y))
{
	waitforpause= WaitForSingleObject(pause, INFINITE);
	CaptureUV(video, U, V);
	Quart(Y, next);
```

After capturing The U and V associated with the Y that we captured inside the while() condition, We prepare the 4×4 down sampled image of the next Y and now we can subtract the pixels of the previous 4×4 down sampled Y pixels from it and threshold them. The next nested loop does that. Note that our threshold is 30.

```
for (int y = 1; y <= R.Y(); y++) {
  for (int x = 1; x <= R.X(); x++) {
       R.B_[y][x] = abs(next.B_[y][x] - previous.B_[y][x]);
       if (R.B_[y][x] > 30)R.B_[y][x] = 255; else R.B_[y][x] = 0;
  }
}
```

Once the subtraction and thresholding of down sampled Y components of current and previous video frames, we can proceed to segment the pixels that are connected. That gives us **THE MOVİNG OBJECTS** in a video.

```
SegmentPixels1(R, seg, map);
```

```
PaintSegments(map, colseg);
```

Next, we convert the YUV image to RGB so that we can display it on to top left corner. We also display the Y component on the top right frame.

```
YUV2RGB32(Y, U, V, RGB);
DisplayImage(F1, RGB);
DisplayImage(F2, Y);
```

This is where we low-pass filter the Y component so that we can apply the Canny edge detector. We need to flip the edge image because the pictures inside the video file is UPSIDE DOWN! Up until now you didn't even notice that the images were upside down because the ICBYTES library has been automatically handling them by putting those images inside a "flipped image." Remember that we have an image type that can be created using the CreateFlippedImage(.) function. The images created with this function are displayed upside down. That is why you haven't notice that all of the images that we dealt with so far were upside down images.

```
Filter_H(Y, edge, filtre, ICB_UCHAR);
Filter_V(edge, filtered, filtre, ICB_UCHAR);
Canny(filtered, edge, 0.2, 0.35);
FlipV(edge, filtered);
DisplayImage(F3, filtered);
edge = 0; filtered = 0;
MagnifyX3(colseg, Mag);
DisplayImage(F4, Mag);
```

Every video has a frame rate. While most of the videos have a frame rate of around 30 frames per second, you may encounter videos with frame rates between 0.2 and 500. So how do we ensure the correct speed of rendering for the video we have? At the beginning of this function, we queried the frame rate of this video and stored it in the "frame_rate" variable. Now, we can use it to calculate how many milliseconds we need to pause between the frames. Obviously, that value is 1000/frame_rate. However, we have done a lot of image processing between displaying each frame which probably took the computer 1 millisecond. That is why we subtract one, and use ((1000/frame_rate)-1) as our pause interval. You can subtract 2 or 3 if your computer is slower.

```
Sleep((1000.0 / frame_rate)-1);
```

We have arrived to the most critical part of our code. Using the **references**, we are switching between the Q1 and Q2 frames and always making sure that they are used one after the other to hold the previous and the next (actually the current) frames. And we do that without spending any processing power because we don't copy any memory.

```
			temp = previous;
			previous = next;
			next = temp;
```

Finally, we release the mutex, in case the user pressed the "PAUSE" button. If that is the case, when we go to the beginning of this loop we shall be stuck until the user presses that button again.

```
			ReleaseMutex(pause);
		}
		return NULL;
}
```

Finally, let's take a look at the code that pauses the rendering:

```
void VideoPause()
{
		static bool paused = false;
		DWORD startpause;
		if (!paused)
		{
			startpause = WaitForSingleObject(pause, INFINITE);
			paused = true;
			ICG_SetWindowText(PAUSEBUTTON, "PLAY");
		}
		else
		{
			ReleaseMutex(pause);
			paused = false;
			ICG_SetWindowText(PAUSEBUTTON, "PAUSE");
		}
}
```

When the first time the user presses this button, it creates a static boolean variable, "paused," and then sets it to false. A static variable inside a function behaves just like a global variable retaining its value even after the function ends. Therefore, when this button is pressed the first time, it goes into the first (upper) part of the if statement and our function holds on to the mutex making the thread function that plays the video to pause because both of them cannot own the mutex. This function also alters the name of the "PAUSE" button using its handle (PAUSEBUTTON) and makes it "PLAY". It also sets the boolean variable "paused" to true.

The next time the user presses this button, whose name has changed to "PLAY", the "paused" variable is true. Therefore, it enters the "else" part of the if statement and releases the mutex letting the thread function to continue rendering the video. The "paused" variable is set to false and the button name is changed back to "PAUSE" again.

Figure 11.3 shows the program paused while playing the “football_cif.y4m” video downloaded from the xiph.org web site. Keep in mind that the techniques described here are not the professional way of doing things. If you want to render videos at commercial quality then you need to learn the thread synchronization techniques in much further detail. For our purposes, the depth of the techniques described in this section are both adequate and simple enough not to confuse the students.

As a practice, students should look for errors and shortcomings of this program and try to fix them.

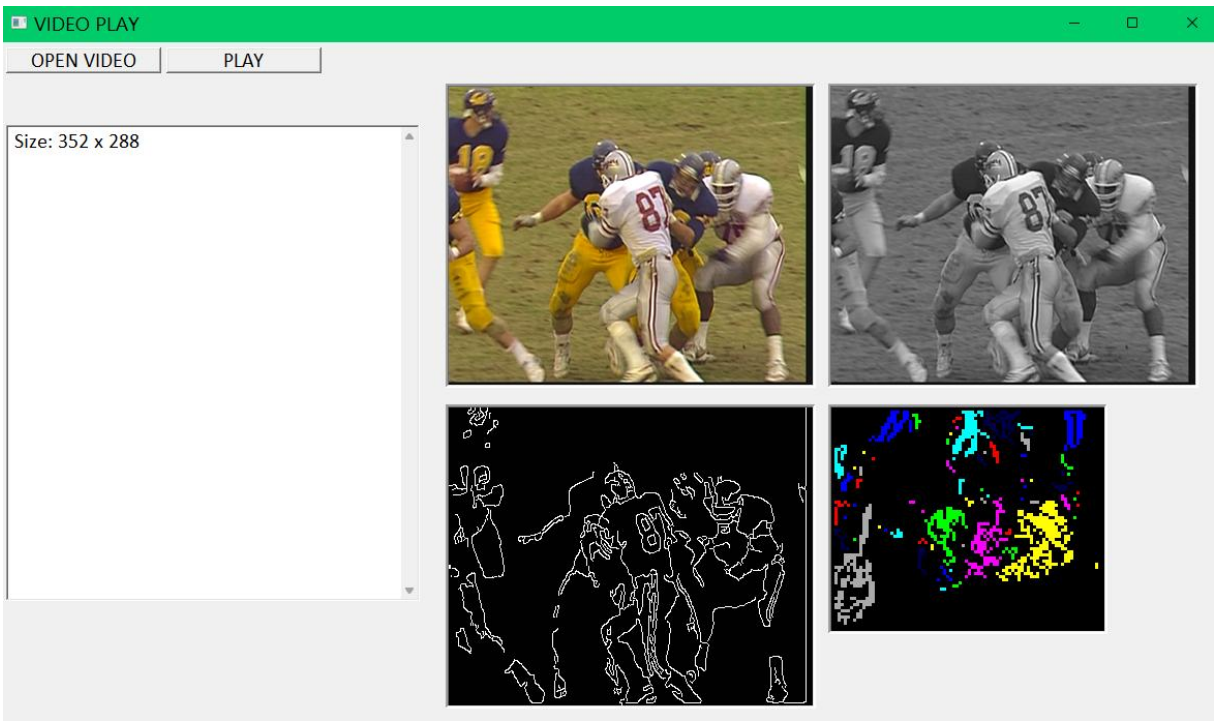


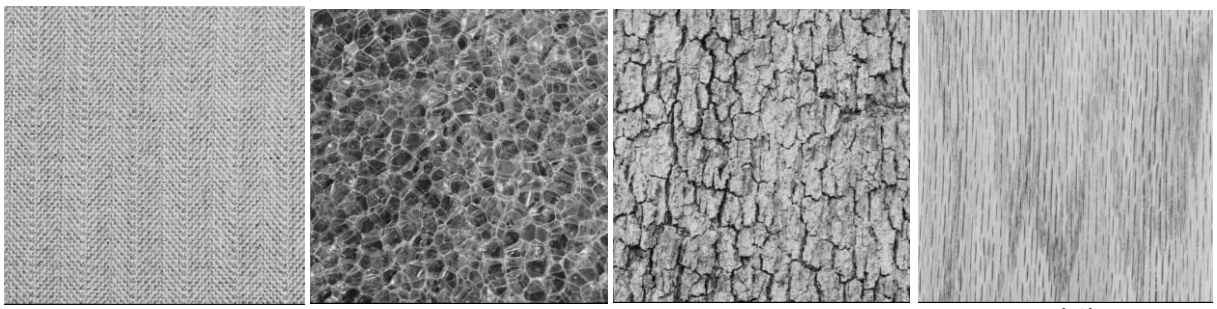
Figure 11.3. The screenshot of the “Video” project while rendering the “football_cif.y4m” video downloaded from the xiph.org web site.

12. Texture Analysis

Consider the pictures shown in Figure 12.1. Our brains can recognize these textures immediately by looking at them. The first one on the left is fabric, the second one is some sort of sponge like plastic bubbles, the third one is a tree bark and the fourth one is wood. We can clearly see repeating patterns in the fabric texture. On the other hand, no plastic bubble is exactly like the other one and the tree bark also has no repeating patterns. If no tree bark texture looks alike, how does our brain know that it is the texture of a tree bark but not a towel or sponge? What is more amazing is that our brains can recognize these textures from any angle or zoom factor. You can look at them closely or from afar and your brain will still recognize them instantly.

Obviously, pixel levels are different for these textures. The plastic bubble image has darker pixels but we know that pixel levels alone cannot define a texture. Even among these textures, the fabric image (a) and the wood image (b) have very similar pixel levels, yet our brain knows that they are different. Textures have randomness in them yet they are distinct in their randomness.

In order to recognize these textures, we need a mathematical theory. If we can't even articulate what the definition of a texture is, how are we going to



(a) (b) (c) (d)

Figure 12.1. Four samples from the famous Brodatz textures. (a) Fabric, (b) Plastic Bubbles, (c) Tree Bark and (d) Wood.

create a mathematical definition? One of the oldest definitions of texture was made by Sklansky, in his article titled "Image segmentation and feature extraction," in IEEE Transactions on Systems, Man, and Cybernetics journal, (SMC-8, 1978, pp 237-247). Unfortunately, even his definition is very vague: A region in an image has a constant texture if a set of local statistics or other local properties of the picture function are constant, slowly varying, or approximately periodic." Because of the fact that a texture is defined in terms of "statistics" and "periodicity," a texture can exist only as a group of pixels, or in other words as a window. There is no point in talking about a single pixel in texture. To define the properties of a texture, we must group pixels and then calculate some sort of statistics.

12.1. The Cooccurrence Matrices

The most famous texture analysis article belongs to Haralick et al., titled "Textural Features for Image Classification," in Transactions on Systems, Man, and Cybernetics journal (in vol. SMC-3, No 6, pp. 610-621, 1973). In that article, a second order pixel statistics is calculated which is called the Gray Level Cooccurrence Matrix or GLCM for short. This statistic is calculated by measuring the probability of a pixel level's occurrence, 'd' pixels away from a given pixel of level 'l'. Figure 12.2 shows these two pixels which are 'd' pixels apart from each other and in four directions that are 45 degrees apart. For example, let's say that your target is pixel at (x=37, y=45) and its value is 166. Your 'd' is 3 and your direction is -45 degree. Then you record the value of pixel at (x=40, y=48). Let's say that pixel's value is 211. Then you go to your matrix, which is 256×256

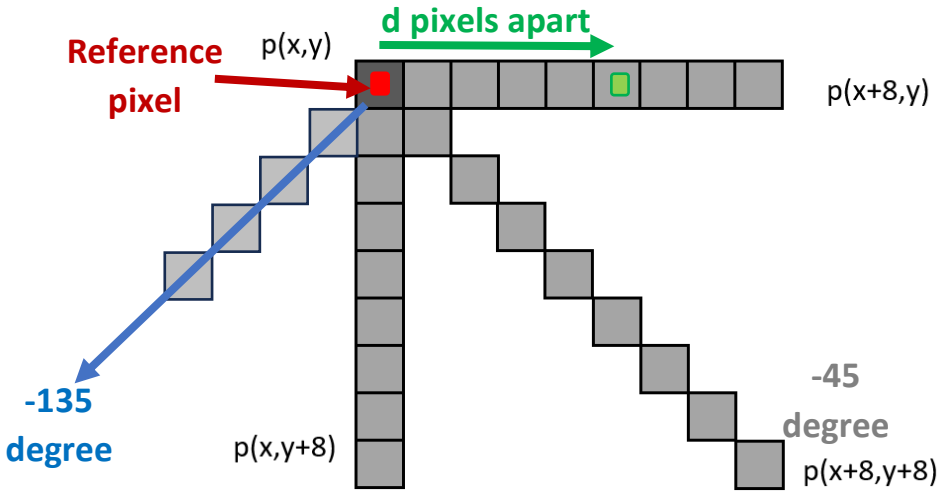


Figure 12.2. Calculation of Gray Level Cooccurrence Matrices in four directions for any pixel that is 'd' pixels away from the target pixel.

elements, and then you increment the element at 166th row and 211th column. If you do that to all of your target pixels or the entire image, you will obtain a second order pixel histogram. If you divide each element of that matrix to the number of your target pixels, then you will obtain a probability matrix which tells you the probability of finding two pixels that are 3 pixels apart diagonally (-45 degree) whose values are A (let's say 166) and B (let's say 211). Surprisingly, scientists discovered that human brain recognizes texture in a very similar way.

In their experiments, Haralick et al. use d=1 and four directions to classify the Brodatz textures. The problem with creating gray level cooccurrence matrices (GLCMs) is that they are too large to use as features. The GLCM is a 256×256 matrix which means that it has 64536 elements. That is for only one direction and d=1. For four directions you will get four 256×256 matrices which is a huge number of elements. In order to reduce these features Haralick derived 14 features that are derived from these matrices. Later on, it was discovered that, out of these 14 features 5 of them really matter and they are:

$$\text{Energy} = \sum_i \sum_j P_d(i,j)^2$$

$$\text{Correlation} = \frac{1}{\sigma_x \sigma_y} \sum_i \left[\sum_j ij P_d(i,j) - \mu_x \mu_y \right]$$

$$\text{Entropy} = -\sum_i \sum_j P_d(i,j) \log[P_d(i,j)]$$

$$\text{Contrast} = \sum_i \sum_j \frac{(i-j)^2 P_d(i,j)}{(N_g - 1)^2}$$

$$\text{Homogeneity} = \sum_i \sum_j \frac{P_d(i,j)}{1 + |i-j|}$$

Where $P_d(i,j)$ are the elements of GLCM after being divided by the number of target pixels, or in other words, the probability matrix. Meanings of the rest of the symbols are:

$$\mu_x = \sum_i i \sum_j P_d(i,j),\ \mu_y = \sum_j j \sum_i P_d(i,j),\ \sigma_x = \sum_i (i-\mu_x)^2 \sum_j P_d(i,j)$$

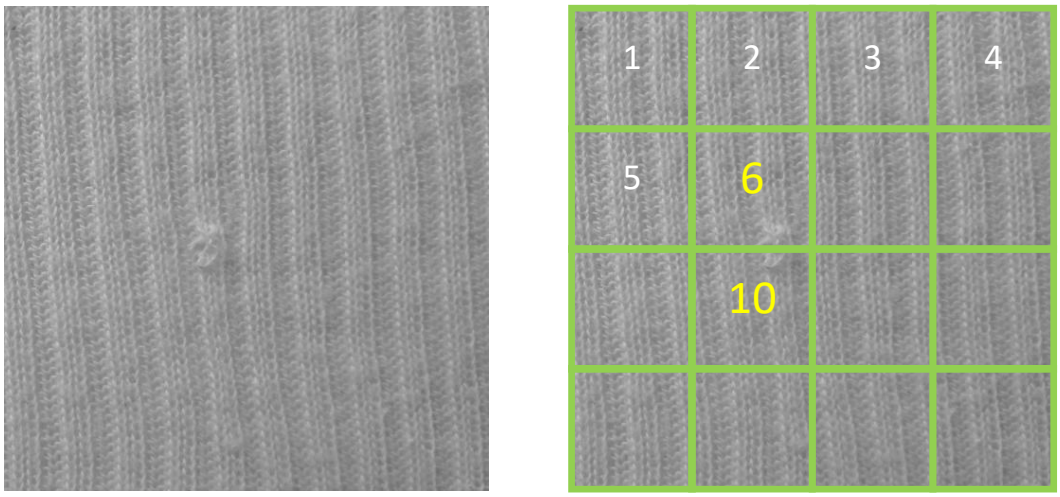


Figure 12.3. A fabric image sample with a weaving defect and the same sample divided into 128×128 pixel wide grid.

The problem here is that calculating these features require us to process a matrix with 65536 elements 8 times which requires huge amount of processing.

If our goal is to distinguish between different types of textures, then we can generate several GLCMs using the entire picture and process only those matrices. However, if our goal is to detect a small defect, which is the common case for quality inspection of fabrics inside a factory, then the statistical fluctuation in the GLCM that a small defect in a large picture creates may be unnoticeable. Or, we may wish to detect not only the existence but also the location of the defect. In that case, we need to divide the image into windows, and generate GLCMs for each window as shown in Figure 12.3. This means that we need to create 16 GLCMs for the sample as shown in Figure 12.3 and then calculate those five features that were listed in the previous page. The amount of data that we need to process became huge, several dozen times the number of pixels in the image itself. And this is only if we use nonoverlapping windows. As you can see from Figure 12.3, the defect is located between window 6 and 10 with most of it on 6. If the statistical variance created by the small portion inside the window 10 is not enough to detect that portion, then we will miss that part. Therefore, a better solution is to use overlapping windows, which almost doubles the huge amount of processing we already had to perform for non-overlapping 16 windows.

What is the solution? The solution is to reduce the gray levels of the image. If we reduce the gray levels of the picture to 16 instead of 256, then instead of

creating 256×256 element GLCMs, we will produce 16×16 GLCMs reducing the amount of data that we need to process to 1/256.

That is why, the popular image processing program, Matlab, has the option to reduce the image gray levels. If you write the following Matlab code and run it, (fabric.bmp image is included with I-See-Bytes library):

```
g=imread('fabric.bmp');
imshow(g)
h=g(1:128,1:128);
gl=graycomatrix(h,'Offset',[0,1],'NumLevels',16);%Reduce gray levels to 16
features=graycoprops(gl)
```

you will get the following output:

```
features =

  struct with fields:

       Contrast: 2.1369
    Correlation: 0.6630
         Energy: 0.0417
    Homogeneity: 0.6146
```

The above Matlab code loads the fabric image that was shown in Figure 12.1-a. And then extracts the first 128×128 portion of that image into the matrix 'h' and then calculates the GLCM in horizontal direction with d=1. Afterwrds, it sends the GLCM to the graycorps(.) function which calculates four of the 14 features that Haralick used in his paper. Note that these 4 features are among 5 features that we recommended two pages ago.

We can calculate the same features using the I-See-Bytes library as well. The following function displays the same features, except for Correlation. At the moment, the I-See-Bytes library calculates a feature that is much faster to calculate, the entropy.

```
void GLCMFeatures()
{
      ICBYTES grey, glcm16,features, rect = { 1,1,128,128 };
      ReadImage("fabric.bmp", grey);
      DisplayImage(FRM1, grey);
      GLCM16(grey, glcm16, rect);
      GLCMFeatures(glcm16, features);
      DisplayMatrix(MLE, features);
}
```

The output of this code is:

(DOUBLE 4 x 1)

2.1369 **3.5305** **0.0417** 0.6146

The first one being the contrast, the second one entropy, the third one energy and the last one homogeneity.

If we go back to the code, we see that different from the Matlab's graycomatrix(.) function, we can send the area of interest as a matrix that contains the upper left corner of the window and its width and height. In Matlab, you need to cut the part of the image that you wish to calculate the GLCM for. But in I-See-Bytes, you can send the coordinates of the window you are interested in. If we go to the definition of GLCM16(.) function inside the ic_pattern.h, we see the following declaration:

```
bool GLCM16(ICBYTES& i, ICBYTES& o, ICBYTES& rect =
*(ICBYTES*)NULL, int direction = 0, int distance = 1);
```

This definition tells us that the "rect" parameter which holds the area of interest(s) is optional. If you don't send it, then the function assumes that you wish to calculate the GLCM for the entire picture. In other words, your area of interest is the entire texture. The "direction" parameter has a default value of zero meaning that you wish to calculate the GLCM in the horizontal (0 degree) direction. That value can be between 0 and 3, corresponding to degrees between 0 and -135 degrees. The distance, d, has a default value of 1, which is the same value that Haralick used in his paper and which is also the most common value for that parameter.

12.2. The Sum and Difference Histograms

Because of the high processing power requirements of the GLCM approach, Michael Unser proposed using Sum and Difference Histograms (SDH). In his article titled "Sum and Difference Histograms for Texture Classification" in the IEEE Trans. Pattern Analysis & Machine Intelligence journal, (Vol.8, No.1 1986, pp. 118-125), he proved that, instead of using second order pixel statistics as the GLCM does, one can use histograms of sum and differences of pixels that are 'd' pixels apart. This approach yields two vectors instead of a matrix, therefore, the amount of data to be processed decreases considerably. A 256 gray level cooccurrence matrix yields 256×256 (64536) element matrix. On the other hand, Unser's method produces two 511 element long vectors (2×511=1022

elements). Furthermore, he proved that most of the information lies in the difference histogram rather than the sum histogram, justifying some researchers such as myself to use only the difference histogram (DH) for the classification of texture.

The DH not only decreases the number of data to be processed, the histogram created using this method is robust against minor changes in the illumination. This can be explained using the This can be explained using the mathematical model of the light absorption. As shown in Figure 12.4, let's assume that a light source is sending photons at an illumination strength of 100. Let the first area corresponding to the pixel-1 has a reflectivity of %80 and the second area corresponding to the pixel-2 has reflectivity of %30. Then the camera would detect the photons coming from these two areas at a strength of 80 and 30 respectively. The sum of these pixels would be 110, and their difference would be 50. If the illumination is increased by %20, then these pixels would be registered as gray levels of 120x0.8=100 and 120x0.3=36. Their sum would be 136 and their difference would be 64. The fluctuation for the difference histogram would be much lower because the increase in the intensities would cancel each other out. If we use 16 gray levels instead of 256, then the difference values for those two different illumination conditions would be 3 and 4. This is only one level shift for %20 change in the illumination which means that for lower illumination changes or for pixels with less contrast difference, it wouldn't

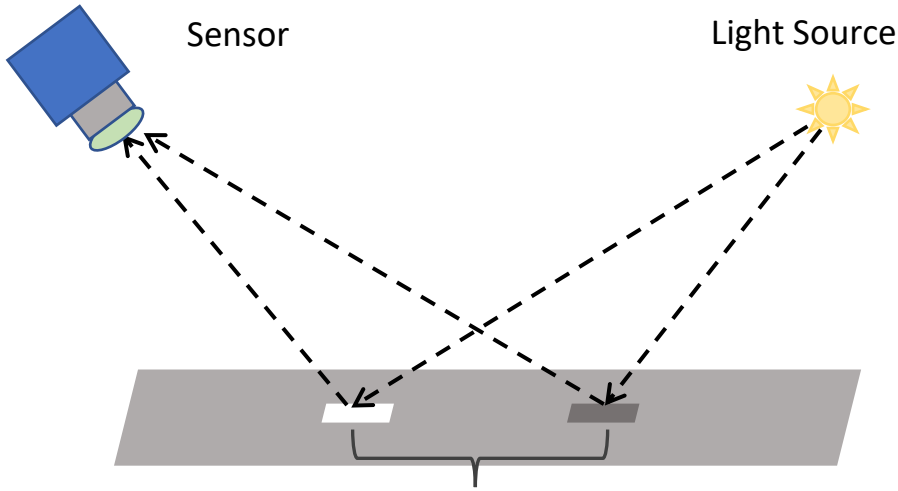


Figure 12.4. Reflection of photons from two different parts of an object. Different absorption rates of the two different regions correspond to different pixel values in the sensor.

be noticeable at all. Nevertheless, applying histogram equalization would be enough for this algorithm to withstand any reasonable illumination fluctuations.

One other benefit of DH is that when the gray levels are reduced to 16, the resultant DH vector has only 31 elements. This so few that, one can use these elements as features and directly feed them to a classifier, instead of calculating the features that Haralick proposed. In the ICBYTES library, the function that calculates the Difference Histograms is called DH16(.) and it has the same input parameters as the GLCM16(.) which was explained in the previous section.